

conference

proceedings

**18th USENIX
Security
Symposium**

Montreal, Canada

August 10–14, 2009

Sponsored by
The USENIX Association

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Thanks to Our Sponsors

BRONZE SPONSOR



Thanks to Our Media Sponsors

acmqueue
Addison-Wesley Professional/
Prentice Hall Professional/
Cisco Press
BetaNews
Conference Guru
Distributed Management Task
Force, Inc.

Free Software Magazine
Hackett and Bankwell:
The Linux Comic
Help Net Security
IEEE Security & Privacy
InfoSec News
Linux Gazette
Linux Journal

Linux Pro Magazine
LXer.com
NetworkWorld ITRoadmap
Conference & Expo
No Starch Press
Toolbox.com
UserFriendly.org

© 2009 by The USENIX Association
All Rights Reserved

This volume is published as a collective work. Rights to individual papers remain with the author or the author's employer. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. USENIX acknowledges all trademarks herein.

ISBN 978-1-931971-69-0

USENIX Association

**Proceedings of the
18th USENIX Security Symposium**

**August 10–14, 2009
Montreal, Canada**

Conference Organizers

Program Chair

Fabian Monrose, *University of North Carolina, Chapel Hill*

Program Committee

Lucas Ballard, *Google Inc.*
Paul Barford, *University of Wisconsin*
Lujo Bauer, *Carnegie Mellon University*
Steven M. Bellovin, *Columbia University*
Bill Cheswick, *AT&T Labs—Research*
George Danezis, *Microsoft Research, UK*
Vinod Ganapathy, *Rutgers University*
Tal Garfinkel, *VMware and Stanford University*
Ian Goldberg, *University of Waterloo*
Trent Jaeger, *Pennsylvania State University*
Samuel King, *University of Illinois at Urbana-Champaign*
Christopher Kruegel, *University of California, Santa Barbara*
Wenke Lee, *Georgia Institute of Technology*
David Lie, *University of Toronto*
Vern Paxson, *UC Berkeley and International Computer Science Institute*
Niels Provos, *Google Inc.*

Michael Reiter, *University of North Carolina, Chapel Hill*

R. Sekar, *Stony Brook University*
Hovav Shacham, *University of California, San Diego*
Micah Sherr, *University of Pennsylvania*
Angelos Stavrou, *George Mason University*
Julie Thorpe, *Carleton University*
Patrick Traynor, *Georgia Institute of Technology*
Wietse Venema, *IBM Research*
David Wagner, *University of California, Berkeley*
Xiaolan (Catherine) Zhang, *IBM Research*

Invited Talks Committee

Dan Boneh, *Stanford University*
Patrick McDaniel, *Pennsylvania State University*
Hugh Thompson, *People Security*

Poster Session Chair

Carrie Gates, *CA Labs*

Work-in-Progress Reports Chair

Sven Dietrich, *Stevens Institute of Technology*

The USENIX Association Staff

External Reviewers

Jonathan Anderson
Haris Andrianakis
Manos Antonakakis
Adam Aviv
Michael Bailey
Arati Baliga
Sandeep Bhatkar
Joseph Bonneau
Kevin Butler
Shakeel Butt
Martim Carbone
Eric Chen
Pau-Chen Cheng
Mihai Christodorescu
Brian Clapper
Sandy Clark
Arel Cordero
Anthony Cozzie
Gabriela Cretu
Eric Cronin
Mohan Dhawan
Artem Dinaburg
William Enck

Adrienne Felt
Matthew Finifter
Deepak Garg
Rosario Gennaro
Chris Grier
Arie Gurfinkel
Matt Hicks
Limin Jia
Rob Johnson
Seny Kamara
David H. King
Stan Kvasov
Lionel Litty
Daniel Luo
Bruce Maggs
Mohammad Mannan
Jonathan McCune
John Mchugh
Steve McLaughlin
Thomas Moyer
Divya Muthukumaran
Deholo Nali
Machigar Ongtang

Bryan Payne
Roberto Perdisci
Moheeb Rajab
Rob Reeder
Richard Reiner
Sandra Rueda
Dave Safford
Amirali Salehi-Abari
Prateek Saxena
Joshua Schiffman
Arvind Seshadri
Gaurav Shah
Monirul Sharif
Kapil Singh
Michael Steiner
Cynthia Sturton
Alok Tongaonkar
Paul Van Oorschot
Werner Vogels
David Whyte
Glenn Wurster
Wei Xu
Junjie Zhang

18th USENIX Security Symposium

August 10–14, 2009

San Jose, CA, USA

Index of Authors	vi
Message from the Program Chair	vii

Wednesday, August 12

Attacks on Privacy

Compromising Electromagnetic Emanations of Wired and Wireless Keyboards.	1
<i>Martin Vuagnoux and Sylvain Pasini, LASEC/EPFL</i>	
Peeping Tom in the Neighborhood: Keystroke Eavesdropping on Multi-User Systems.	17
<i>Kehuan Zhang and XiaoFeng Wang, Indiana University, Bloomington</i>	
A Practical Congestion Attack on Tor Using Long Paths	33
<i>Nathan S. Evans, University of Denver; Roger Dingledine, The Tor Project; Christian Grothoff, University of Denver</i>	

Memory Safety

Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors	51
<i>Periklis Akrkitidis, Computer Laboratory, University of Cambridge; Manuel Costa and Miguel Castro, Microsoft Research, Cambridge; Steven Hand, Computer Laboratory, University of Cambridge</i>	
Dynamic Test Generation to Find Integer Bugs in x86 Binary Linux Programs	67
<i>David Molnar, Xue Cong Li, and David A. Wagner, University of California, Berkeley</i>	
Memory Safety for Low-Level Software/Hardware Interactions	83
<i>John Criswell, University of Illinois; Nicolas Geoffray, Université Pierre et Marie Curie, INRIA/Regal; Vikram Adve, University of Illinois</i>	

Network Security

Detecting Spammers with SNARE: Spatio-temporal Network-level Automatic Reputation Engine.	101
<i>Shuang Hao, Nadeem Ahmed Syed, Nick Feamster, and Alexander G. Gray, Georgia Tech; Sven Krasser, McAfee, Inc.</i>	
Improving Tor using a TCP-over-DTLS Tunnel	119
<i>Joel Reardon, Google Switzerland GmbH; Ian Goldberg, University of Waterloo</i>	
Locating Prefix Hijackers using LOCK.	135
<i>Tongqing Qiu, Georgia Tech; Lusheng Ji, Dan Pei, and Jia Wang, AT&T Labs—Research; Jun (Jim) Xu, Georgia Tech; Hitesh Ballani, Cornell University</i>	

Thursday, August 13

JavaScript Security

GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code	151
<i>Salvatore Guarnieri, University of Washington; Benjamin Livshits, Microsoft Research</i>	
NOZZLE: A Defense Against Heap-spraying Code Injection Attacks	169
<i>Paruj Ratanaworabhan, Cornell University; Benjamin Livshits and Benjamin Zorn, Microsoft Research</i>	
Cross-Origin JavaScript Capability Leaks: Detection, Exploitation, and Defense	187
<i>Adam Barth, Joel Weinberger, and Dawn Song, University of California, Berkeley</i>	

Radio

Physical-layer Identification of RFID Devices	199
<i>Boris Danev, ETH Zürich, Switzerland; Thomas S. Heydt-Benjamin, IBM Zürich Research Laboratory, Switzerland; Srdjan Capkun, ETH Zürich, Switzerland</i>	
CCCP: Secure Remote Storage for Computational RFIDs	215
<i>Mastooreh Salajegheh, Shane Clark, Benjamin Ransford, and Kevin Fu, University of Massachusetts Amherst; Ari Juels, RSA Laboratories, The Security Division of EMC</i>	
Jamming-resistant Broadcast Communication without Shared Keys	231
<i>Christina Pöpper, Mario Strasser, and Srdjan Capkun, ETH Zürich, Switzerland</i>	

Securing Web Apps

xBook: Redesigning Privacy Control in Social Networking Platforms	249
<i>Kapil Singh, Georgia Institute of Technology; Sumeer Bhola, Google; Wenke Lee, Georgia Institute of Technology</i>	
Nemesis: Preventing Authentication & Access Control Vulnerabilities in Web Applications	267
<i>Michael Dalton and Christos Kozyrakis, Stanford University; Nickolai Zeldovich, CSAIL, MIT</i>	
Static Enforcement of Web Application Integrity Through Strong Typing.	283
<i>William Robertson and Giovanni Vigna, University of California, Santa Barbara</i>	

Applied Crypto

Vanish: Increasing Data Privacy with Self-Destructing Data	299
<i>Roxana Geambasu, Tadayoshi Kohno, Amit A. Levy, and Henry M. Levy, University of Washington</i>	
Efficient Data Structures For Tamper-Evident Logging	317
<i>Scott A. Crosby and Dan S. Wallach, Rice University</i>	
VPriv: Protecting Privacy in Location-Based Vehicular Services	335
<i>Raluca Ada Popa and Hari Balakrishnan, Massachusetts Institute of Technology; Andrew J. Blumberg, Stanford University</i>	

Friday, August 14

Malware Detection and Protection

- Effective and Efficient Malware Detection at the End Host 351
Clemens Kolbitsch and Paolo Milani Comparetti, Secure Systems Lab, TU Vienna; Christopher Kruegel, University of California, Santa Barbara; Engin Kirda, Institute Eurecom, Sophia Antipolis; Xiaoyong Zhou and XiaoFeng Wang, Indiana University at Bloomington
- Protecting Confidential Data on Personal Computers with Storage Capsules 367
Kevin Borders, Eric Vander Weele, Billy Lau, and Atul Prakash, University of Michigan
- Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms 383
Ralf Hund, Thorsten Holz, and Felix C. Freiling, Laboratory for Dependable Distributed Systems, University of Mannheim, Germany

Browser Security

- Crying Wolf: An Empirical Study of SSL Warning Effectiveness 399
Joshua Sunshine, Serge Egelman, Hazim Almuhtimedi, Neha Atri, and Lorrie Faith Cranor, Carnegie Mellon University
- The Multi-Principal OS Construction of the Gazelle Web Browser 417
Helen J. Wang, Microsoft Research; Chris Grier, University of Illinois at Urbana-Champaign; Alex Moshchuk, University of Washington; Samuel T. King, University of Illinois at Urbana-Champaign; Piali Choudhury and Herman Venter, Microsoft Research

Index of Authors

Adve, Vikram, 83
Ahmed Syed, Nadeem, 101
Akritidis, Periklis, 51
Almuhimedi, Hazim, 399
Atri, Neha, 399
Balakrishnan, Hari, 335
Ballani, Hitesh, 135
Barth, Adam, 187
Bhola, Sumeer, 249
Blumberg, Andrew J., 335
Borders, Kevin, 367
Capkun, Srdjan, 199, 231
Castro, Miguel, 51
Choudhury, Piali, 417
Clark, Shane, 215
Costa, Manuel, 51
Cranor, Lorrie Faith, 399
Criswell, John, 83
Crosby, Scott A., 317
Dalton, Michael, 267
Danev, Boris, 199
Dingledine, Roger, 33
Egelman, Serge, 399
Evans, Nathan S., 33
Feamster, Nick, 101
Freiling, Felix C., 383
Fu, Kevin, 215
Geambasu, Roxana, 299
Geoffray, Nicolas, 83
Goldberg, Ian, 119
Gray, Alexander G., 101
Grier, Chris, 417
Grothoff, Christian, 33
Guarnieri, Salvatore, 151
Hand, Steven, 51
Hao, Shuang, 101
Heydt-Benjamin, Thomas S., 199
Holz, Thorsten, 383
Hund, Ralf, 383
Ji, Lusheng, 135
Juels, Ari, 215
King, Samuel T., 417
Kirda, Engin, 351
Kohno, Tadayoshi, 299
Kolbitsch, Clemens, 351
Kozyrakis, Christos, 267
Krasser, Sven, 101
Kruegel, Christopher, 351
Lau, Billy, 367
Lee, Wenke, 249
Levy, Amit A., 299
Levy, Henry M., 299
Li, Xue Cong, 67
Livshits, Benjamin, 151, 169
Milani Comparetti, Paolo, 351
Molnar, David, 67
Moshchuk, Alex, 417
Pasini, Sylvain, 1
Pei, Dan, 135
Popa, Raluca Ada, 335
Pöpper, Christina, 231
Prakash, Atul, 367
Qiu, Tongqing, 135
Ransford, Benjamin, 215
Ratanaworabhan, Paruj, 169
Reardon, Joel, 119
Robertson, William, 283
Salajegheh, Mastrooreh, 215
Singh, Kapil, 249
Song, Dawn, 187
Strasser, Mario, 231
Sunshine, Joshua, 399
Vander Weele, Eric, 367
Venter, Herman, 417
Vigna, Giovanni, 283
Vuagnoux, Martin, 1
Wagner, David A., 67
Wallach, Dan S., 317
Wang, Helen J., 417
Wang, Jia, 135
Wang, XiaoFeng, 17, 351
Weinberger, Joel, 187
Xu, Jun (Jim), 135
Zeldovich, Nickolai, 267
Zhang, Kehuan, 17
Zhou, Xiaoyong, 351
Zorn, Benjamin, 169

Message from the Program Chair

Dear colleagues,

It is my pleasure to welcome you to the 18th USENIX Security Symposium, held in beautiful Montreal, Canada. I am delighted to report that this year's program continues the tradition of excellence established over the long and prestigious history of the conference. The 176 submissions we received represent one of the most successful calls in the history of the conference. Of the papers submitted to the technical program, one was later withdrawn by the authors, and three were double submissions that were summarily rejected. The remaining 172 submissions underwent a thorough reviewing process, and it was exceedingly difficult to select only 26 papers from such a high number of strong submissions spanning many areas in computer security. At such a selective acceptance rate (i.e., 15%), many of the papers that we could not accept were also of high quality; surely some of these works will be accepted at other top-tier conferences in the near future. I sincerely hope that the authors who submitted papers found the reviewers' feedback helpful.

The reviewing process began a few days after the submissions deadline, and every paper was assigned to at least three reviewers on the Program Committee (PC). Several papers received as many as five reviews. The papers were evaluated based on scientific novelty, contribution to the field, and technical quality. Our Program Committee meeting was held April 6–7 at the University of North Carolina, Chapel Hill. In keeping with the strong USENIX Security tradition, all 26 committee members attended the PC meeting. The top 62 papers, having made it through a lengthy on-line discussion phase, were further deliberated on at the meeting, and the final papers were selected based on input from the entire committee. It was truly an honor to work with such a dedicated and collegial team of subject matter experts! The service of all members of the PC was remarkable, and I would like to personally thank each of them for their outstanding service to the community. I am also grateful to the many external reviewers (acknowledged in the frontmatter) who assisted the committee members.

This year, we are also very fortunate to have an exceptional lineup of events that broaden the technical program. I would like to thank Dan Boneh, Patrick McDaniel, and Hugh Thompson for their tireless efforts to produce the Invited Talks program. Thanks also to Sven Dietrich for coordinating the always entertaining Work-in-Progress (WiPs) reports, and to Carrie Gates for agreeing to build on last year's success by again organizing the Poster Session. I am sure you will find the Symposium an exciting and rewarding event.

Last but not least, I would like to thank the USENIX staff for their support throughout the entire process. What an outstanding team! Without a doubt, this process would not have run as smoothly were it not for their dedication. I am indebted to Anton Cohen, Anne Dickison, Casey Henderson, Jane-Ellen Long, and Devon Shaw for their patience and support. And, of course, my deepest gratitude goes to the masterful organizer, Ellie Young, who seamlessly ran the entire show behind the scenes. (Who knows? I might even reconsider going through this grueling, yet rewarding, process again just because of her—but don't hold me to that!) Thanks also to the liaisons to the USENIX Board for providing guidance on numerous issues that surfaced during the selection process.

Again, welcome to the conference. We hope you enjoy the program as much as we enjoyed putting it together!

Fabian Monrose, University of North Carolina at Chapel Hill
Program Chair

Compromising Electromagnetic Emanations of Wired and Wireless Keyboards

Martin Vuagnoux
LASEC/EPFL
martin.vuagnoux@epfl.ch

Sylvain Pasini
LASEC/EPFL
sylvain.pasini@epfl.ch

Abstract

Computer keyboards are often used to transmit confidential data such as passwords. Since they contain electronic components, keyboards eventually emit electromagnetic waves. These emanations could reveal sensitive information such as keystrokes. The technique generally used to detect compromising emanations is based on a wide-band receiver, tuned on a specific frequency. However, this method may not be optimal since a significant amount of information is lost during the signal acquisition. Our approach is to acquire the raw signal directly from the antenna and to process the entire captured electromagnetic spectrum. Thanks to this method, we detected four different kinds of compromising electromagnetic emanations generated by wired and wireless keyboards. These emissions lead to a full or a partial recovery of the keystrokes. We implemented these side-channel attacks and our best practical attack fully recovered 95% of the keystrokes of a PS/2 keyboard at a distance up to 20 meters, even through walls. We tested 12 different keyboard models bought between 2001 and 2008 (PS/2, USB, wireless and laptop). They are all vulnerable to at least one of the four attacks. We conclude that most of modern computer keyboards generate compromising emanations (mainly because of the manufacturer cost pressures in the design). Hence, they are not safe to transmit confidential information.

1 Introduction

Today, most of the practical attacks on computers exploit software vulnerabilities. New security weaknesses are disclosed every day, but patches are commonly delivered within a few days. When a vulnerability is based on hardware, there is generally no software update to avoid the exposure: the device must be changed.

Computer keyboards are often used to transmit sensitive information such as passwords, e.g. to log into com-

puters, to do e-banking money transfer, etc. A weakness in these hardware devices will jeopardize the security of any password-based authentication system.

Compromising electromagnetic emanation problems appeared already at the end of the 19th century. Because of the extensive use of telephones, wire networks became extremely dense. People could sometimes hear other conversations on their phone line due to undesired coupling between parallel wires. This unattended phenomenon, called *crosstalk*, may be easily canceled by twisting the cables.

A description of some early exploitations of compromising emanations has been recently declassified by the National Security Agency [26]. During World War II, the American Army used teletypewriter communications encrypted with Bell 131-B2 mixing devices. In a Bell laboratory, a researcher noticed, quite by accident, that each time the machine stepped, a spike appeared on an oscilloscope in a distant part of the lab. To prove the vulnerability of the device, Bell engineers captured the compromising emanations emitted by a Bell 131-B2, placed in a building across the street and about 25 meters away. They were able to recover 75% of the plaintext.

During the Vietnam war, a sensor called *Black Crow* carried aboard C-130 gunships was able to detect the electromagnetic emanations produced by the ignition system of trucks on the Ho Chi Minh trail, from a distance up to 10 miles [25, 11].

1.1 Related Work

Academic research on compromising electromagnetic emanations started in the mid 1980's and there has been significant recent progresses [28, 1]. The threat related to compromising emanations has been constantly confirmed by practical attacks such as Cathode Ray Tubes (CRT) displays image recovery [34], Liquid Crystal Display (LCD) image recovery [20], secret key disclosure [16], video displays risks [18, 33] or radiations from

FPGAs [24].

Compromising electromagnetic emanations of serial-port cables have been already discussed by Smulders [30] in 1990. PS/2 keyboards still use bi-directional serial communication to transmit the pressed key code to the computer. Hence, some direct compromising electromagnetic emanations might appear. However, the characteristics of the serial line changed since the 90's. The voltage is not 15 volts anymore and the transition times of the signals are much longer (from picoseconds to microseconds).

Since keyboards are often the first input device of a computer system, they have been intensively studied. For instance, the exploitation of visual compromising information leaks such as optical reflections [5] which could be applied to keyboards, the analysis of surveillance video sequences [6] which can be used by an attacker to recover the keystrokes (even with a simple webcam) or the use of the blinking LEDs of the keyboard as a covert channel [21]. Acoustic compromising emanations from keyboards have been studied as well. Asonov and Agrawal [4] discovered that each keystroke produces a unique sound when it is pressed or released and they presented a method to recover typed keystrokes with a microphone. This attack was later improved, see [38, 7]. Even passive timing analysis may be used to recover keystrokes. Song et al. highlighted that the keystroke timing data measured in older SSH implementations [32] may be used to recover encrypted passwords. A risk of compromising emission from keyboards has been postulated by Kuhn and Anderson [20, 17, 2]. They also proposed countermeasures (see US patent [3]). Some unofficial documents on *TEMPEST* [37] often designate keyboards as potential information leaking devices. However, we did not find any experiment or evidence proving or refuting the practical feasibility to remotely eavesdrop keystrokes, especially on modern keyboards.

1.2 Our Contribution

This paper makes the following main contributions:

A Full Spectrum Acquisition Method. To detect compromising electromagnetic emanations a receiver tuned on a specific frequency is generally used. It brings the signal in base band with a limited bandwidth. Therefore, the signal can be demodulated in amplitude (AM) or frequency (FM). This method might not be optimal. Indeed, the signal does not contain the maximal entropy since a significant amount of information is lost. We propose another approach. We acquire the raw signal directly from the antenna and analyze the entire captured electromagnetic spectrum with Short Time Fourier Transform (also known as *Waterfall*) to distill potential compromising emanations.

The Study of Four Different Sources of Information Leakage from Keyboards. To determine if keyboards generate compromising emanations, we measured the electromagnetic radiations emitted when a key is pressed. Due to our improved acquisition method, we discovered several direct and indirect compromising emanations which leak information on the keystrokes. The first technique looks at the emanations of the falling edges (i.e. the transition from a high logic state to a low logic state) from the bi-directional serial cable used in the PS/2 protocol. It can be used to reveal keystrokes with about 1 bit of uncertainty. The second approach uses the same source, but consider the rising and the falling edges of the signal to recover the keystrokes with 0 bits of uncertainty. The third approach is focused on the harmonics emitted by the keyboard to recover the keystrokes with 0 bits of uncertainty. The last approach considers the emanations emitted from the matrix scan routine (used by PS/2, USB and Wireless keyboards) and yields about 2.5 bits of uncertainty per keystroke. This compromising emanation has been previously posited by Kuhn and Anderson [3], although that work provided no detailed analysis.

The Implementation and the Analysis of Four Keystroke Recovery Techniques in Four Different Scenarios. We tested 12 different keyboard models, with PS/2, USB connectors and wireless communication in different setups: a semi-anechoic chamber, a small office, an adjacent office and a flat in a building. We demonstrate that these keyboards are all vulnerable to at least one of the four keystroke recovery techniques in all scenarios. The best attack successfully recovers 95% of the keystrokes at a distance up to 20 meters, even through walls. Because each keyboard has a specific *fingerprint* based on the clock frequency inconsistencies, we can determine the source keyboard of a compromising emanation, even if multiple keyboards from the same model are used at the same time. First, we did the measurements in a semi-anechoic electromagnetic chamber to isolate the device from external noise. Then we confirmed that these compromising emanations are exploitable in real situations.

We conclude that most of modern computer keyboards generate compromising emanations (mainly because of the manufacturer cost pressures in the design). Hence they are not safe to transmit confidential information.

1.3 Structure of the Paper

Section 2 describes some basics on compromising electromagnetic emanations. In Section 3 we present our acquisition method based on Short Time Fourier Transform. In Section 4 we present four different setups used for the measurements, from a semi-anechoic chamber to

real environments. In Section 5 we give the complete procedure used to detect the compromising electromagnetic emanations. Then, we detail the four different techniques. In Section 6, we give the results of our measurements in different setups. In Section 7, we describe some countermeasures to avoid these attacks. In Section 8, we give some extensions and improvements. Finally we conclude.

2 Electromagnetic Emanations

Electromagnetic compatibility (EMC) is the analysis of electromagnetic interferences (EMI) or Radio Frequency Interferences (RFI) related to electric devices. EMC aims at reducing unintentional generation, propagation and reception of electromagnetic energy in electric systems. EMC defines two kinds of unwanted emissions: conductive coupling and radiative coupling. Conductive coupling requires physical support such as electric wires to transmit interferences through the system. Radiative coupling occurs when a part of the internal circuit acts as an antenna and transmits undesired electromagnetic waves. EMC generally distinguishes two types of electromagnetic emissions depending on the kind of the radiation source: differential-mode and common-mode.

Differential-mode radiation is generated by loops formed by components, printed circuit traces, ribbon cables, etc. These loops act as small circular antennas and eventually radiate. These radiations are generally low and do not disturb the whole system. Differential-mode signals are not easily influenced by external radiations. Moreover they can be easily avoided by shielding the system.

Common-mode radiation is the result of undesired internal voltage drops in the circuit which usually appear in the ground loop. Indeed, ground loop currents are due to the unbalanced nature of ordinary transmitting and receiving circuits. Thus, external cables included in the ground loop act as antennas excited by some internal voltages. Because these voltage drops are not intentionally created by the system, it is generally harder to detect and control common-mode radiations than differential-mode radiations.

From the attacker's point of view there are two types of compromising emanations: direct and indirect emanations.

Direct Emanations. In digital devices, data is encoded with logic states, generally described by short burst of square waves with sharp rising and falling edges. During the transition time between two states, electromagnetic waves are eventually emitted at a maximum frequency related to the duration of the rise/fall time. Because these compromising radiations are provided straight by

the wire transmitting sensitive data, they are called direct emanations.

Indirect Emanations. Electromagnetic emanations may interact with active electronic components which induce new types of radiations. These unintended emanations manifest themselves as modulations or inter-modulations (phase, amplitude or frequency) or as carrier signals e.g. clock and its harmonics. Non-linear coupling between carrier signals and sensitive data signals, crosstalk, ground pollution or power supply DC pollution may generate compromising modulated signals. These indirect emanations may have better propagation than direct emanations. Hence, they may be captured at a larger range. The prediction of these emanations is extremely difficult. They are generally discovered during compliance tests such as FCC [15], CISPR [10], MIL-STD-461 [22], NACSIM-5000 [37], etc.

3 Electromagnetic Signal Acquisition

Two techniques are generally used to discover compromising electromagnetic emanations.

3.1 Standard Techniques

A method consists in using a spectral analyzer to detect signal carriers. Such a signal can be caught only if the duration of the carrier is significant. This makes compromising emanations composed of peaks difficult to detect with spectral analyzers.

Another method is based on a wide-band receiver tuned on a specific frequency. Signal detection process consists in scanning the whole frequency range of the receiver and to demodulate the signal according to its amplitude modulation (AM) or frequency modulation (FM). When an interesting frequency is discovered, narrow-band antennas and some filters are used to improve the Signal-to-Noise Ratio (SNR) of the compromising emanations. In practice, wide-band receivers such as R-1250 [19] and R-1550 [12] from Dynamic Sciences International, Inc. are used, see [17, 1]. Indeed, these receivers are compliant with secret requirements for the NACSIM-5000 [37] also known as *TEMPEST*. These devices are quite expensive and unfortunately not owned by our lab. Hence, we used a cheaper and open-source solution based on the USRP (Universal Software Radio Peripheral) [14] and the GNU Radio project [35]. The USRP is a device which allows you to create a software radio using any computer with USB port. With different daughterboards, the USRP is able to scan from DC to 2.9 GHz with a sample rate of 64 MS/s at a resolution of 12 bits. The full range on the ADC is 2 volts peak to peak and the input is 50 ohms differential. The GNU

Radio project is a powerful software library used by the USRP to process various modulations (AM, FM, PSK, FSK, etc.) and signal processing constructs (optimized filters, FFT, etc.). Thus, the USRP and the GNU Radio project may act as a wide-band receiver and a spectral analyzer with software-based FFT computation.

3.2 Novel Techniques

Some direct and indirect electromagnetic emanations may stay undetected with standard techniques, especially if the signal is composed of irregular peaks or erratic frequency carriers. Indeed, spectral analyzers need significantly static carrier signals. Similarly, the scanning process of wide-band receivers is not instantaneous and needs a lot of time to cover the whole frequency range. Moreover the demodulation process may hide some interesting compromising emanations.

In this paper, we use a different method to detect compromising electromagnetic emanations of keyboards. First, we obtain the raw signal directly from the antenna instead of a filtered and demodulated signal with limited bandwidth. Then, we compute the Short Time Fourier Transform (STFT), which gives a 3D signal with time, frequency and amplitude.

Modern analog-to-digital converters (ADC) provide very high sampling rates (Giga samples per second). If we connect an ADC directly to a wide-band antenna, we can import the raw sampled signal to a computer and we can use software radio libraries to instantly highlight potentially compromising emanations. The STFT computation of the raw signal reveals the carriers and the peaks even if they are present only for a short time.

Unfortunately there is no solution to transfer the high amount of data to a computer in real time. The data rate is too high for USB 2.0, IEEE 1394, Gigabit Ethernet or Serial ATA (SATA) interfaces. However, with some smart triggers, we can sample only the (small) interesting part of the signal and we store it in a fast access memory. Oscilloscopes provide triggered analog-to-digital converters with fast memory. We used a Tektronix TDS5104 with 1 Mpt memory and a sample rate of 5 GS/s. It can acquire electromagnetic emanations up to 2.5 GHz according to the Nyquist theorem. Moreover, this oscilloscope has antialiasing filters and supports IEEE 488 General Purpose Interface Bus (GPIB) communications. We developed a tool to define some specific triggers (essentially peak detectors) and to export the acquired data to a computer under GNU/Linux over Ethernet. Thus the signal can be processed with the GNU Radio software library and some powerful tools such as Baudline [29] or the GNU project Octave [13]. The advantage of this method is to process the raw signal, which is directly sampled from the antenna without

any demodulation. Moreover, all compromising electromagnetic emanations up to a frequency of 2.5 GHz are captured. Thus, with this technique, we are able to highlight compromising emanations quickly and easily. This solution is ideal for very short data burst transmissions used by computer keyboards.

4 Experimental Setup

The objective of this experiment is to observe the existence of compromising emanations of computer keyboards when a key is pressed. Obviously electromagnetic emanations depend on the environment. We defined four different setups.

Setup 1: The Semi-Anechoic Chamber. We used a professional semi-anechoic chamber (7×7 meters). Our aim was not to cancel signal echos but to avoid external electromagnetic pollution (Faraday cage). The antenna was placed up to 5 meters from the keyboard connected to a computer (the maximum distance according to the echo isolation of the room). The tested keyboard was on a one meter high table and the computer (PC tower) was on the ground.

Setup 2: The Office. To give evidence of the feasibility of the attacks with background noise, we measured the compromising emanations of the keyboards in a small office (3×5 meters) with two powered computers and three LCD displays. The electromagnetic background noise was quite important with a cluster of 40 computers 10 meters away from the office, more than 60 powered computers on the same floor and a 802.11n wireless router at less than 3 meters away from the office. The antenna was in the office and moved back through the opened door up to 10 meters away from the keyboard in order to determine the maximum range.

Setup 3: The Adjacent Office. This setup is similar to the office setup but we measured the compromising emanations of the keyboards from an adjacent office through a wall of 15 cm composed of wood and plaster.

Setup 4: The Building. This setup takes place in a flat which is in a building of five floors in the center of a mid-size city. The keyboard was in the fifth floor. We performed measurements with the antenna placed on the same floor first. Then, we moved the antenna as far as the basement (up to 20 meter from the keyboard).

Antennas. Since the compromising emanations were found on frequency bands between 25 MHz and 300 MHz, we used a biconical antenna (50 Ohms VHA 9103 Dipol Balun) to improve the Signal-to-Noise Ratio (SNR). We also tested if these compromising emanations can be captured with a smaller antenna such as a simple loop made of a wire of copper (one meter long).

Keyboards. We picked 12 different keyboard models present in our lab: 7 PS/2 keyboards (Keyboard A1-A7), 2 USB keyboards (Keyboard B1-B2), 2 Laptop keyboards (Keyboard C1-C2) and 1 wireless keyboard (Keyboard D1). They were all bought between 2001 and 2008. We also collected measurements with the keyboard connected to a laptop with battery to avoid possible conductive coupling through the power supply. For obvious security reasons, we do not give the brand name and the model of the tested keyboards.

5 Discovering and Exploiting Emanations

To discover compromising emanations, we placed Keyboard A1 in the semi-anechoic chamber and we used the biconical antenna. A diagram of the experiment is depicted in Figure 1. We acquired the raw signal with the oscilloscope as explained above. Since the memory of the oscilloscope is limited, we have to precisely trigger data acquisition. First, we used the earliest falling edge of the data signal sent when a key is pressed. We physically connected a probe on the data wire of the cable between the keyboard and the computer.

Figure 2 gives the STFT of the captured raw signal when the key E is pressed on an American keyboard. With only one capture we are able to represent the entire spectrum along the full acquisition time. In addition, we have a visual description of all electromagnetic emanations. In particular we clearly see some carriers (vertical lines) and broadband impulses (horizontal lines). The three first techniques are based on these compromising emanations and are detailed in the following sections.

Our objective is to use an electromagnetic trigger, since we normally do not have access to the data wire. The discovered broadband impulses (horizontal lines) can be used as a trigger. Thus, with only an antenna, we are able to trigger the acquisition of the compromising electromagnetic emanations. More details are given below.

Some keyboards do not emit electromagnetic emanations when a key is pressed. But with a different trigger model, based on peak detector as well, we discovered another kind of emission, continuously generated (even if no key is pressed). This is the last technique, detailed in Section 5.4.

5.1 The Falling Edge Transition Technique

To understand how direct compromising emanations may be generated by keyboards, we need to briefly describe the PS/2 communication protocol. According to [9], when a key is pressed, released or held down, the keyboard sends a packet of information known as a

scan code to the computer. In the default scan code set¹, most of the keys are one-byte long encoded. Some extended keys are two or more bytes long. These codes can be identified by the fact that their first byte is 0xE0. The protocol used to transmit these scan codes is a bi-directional serial communication, based on four wires: Vcc (5 volts), ground, data and clock. For each byte of the scan code, the keyboard pulls down the clock signal at a frequency between 10 KHz and 16.7 KHz for 11 clock cycles. When the clock is low, the state of the signal data is read by the computer. The 11 bits sent correspond to a start bit (0), 8 bits for the scan code of the pressed key (least significant bit first), an odd parity check bit on the byte of the scan code (the bit is set if there is an even number of 1's), and finally a stop bit (1). Figure 3 represents both data and clock signals when the key E is pressed. Note that the scan code is binded to

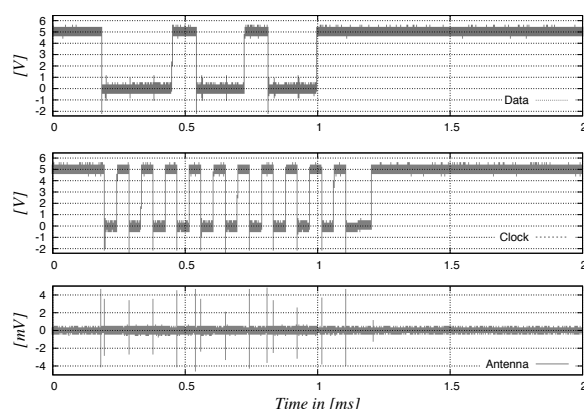


Figure 3: Data, clock and the compromising emanation captured (semi-anechoic chamber, Keyboard A1) with the loop antenna at 5 meters (a wire of copper, one meter long) when the key E (0x24) is pressed. Data signal sends the message: 0 00100100 1 1.

a physical button on the keyboard, it does not represent the character printed on that key. For instance, the scan code of E is 0x24 if we consider the American layout keyboard.

Logic states given by data and clock signals in the keyboard are usually generated by an open collector coupled to a pull-up resistor. The particularity of this system is that the duration of the rising edge is significantly longer (2 μ s) than the duration of the falling edge (200 ns). Thus, the compromising emanation of a falling edge should be much more powerful (and with a higher maximum frequency) than the rising edge. This property is known and has been already noticed by Kuhn [17, p.35]. Clock and data signals are identically generated. Hence,

¹There are three different scan code sets, but the second one is commonly used by default.

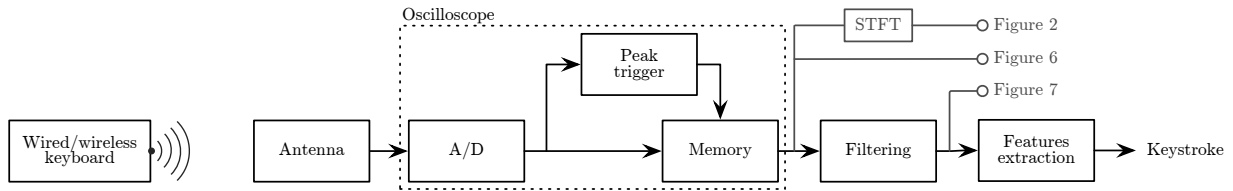


Figure 1: Diagram of our equipment for the experiments.

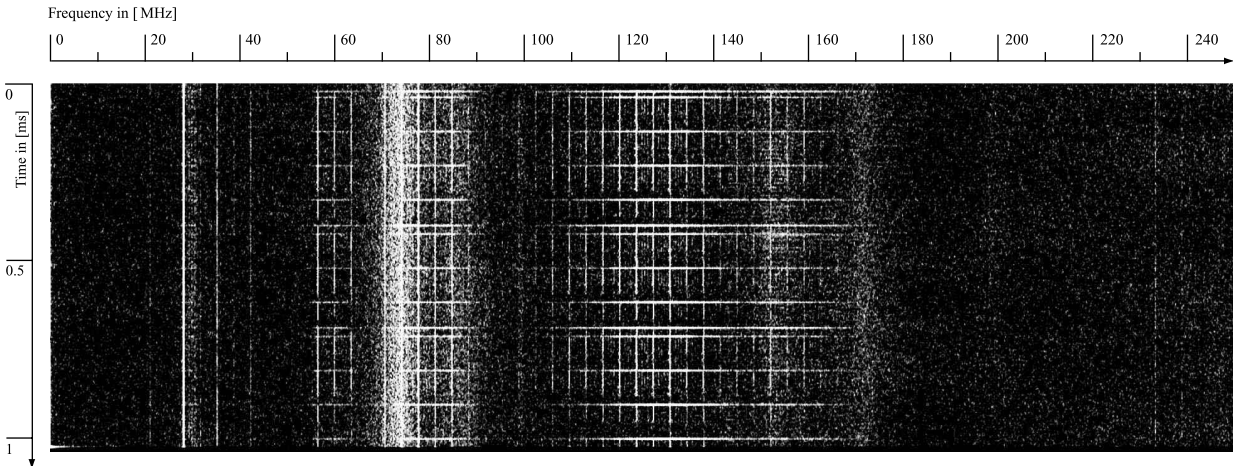


Figure 2: Short Time Fourier Transform (STFT) of the raw signal depicted in Figure 6 (Kaiser windowing of 40, 65536 points)

the compromising emanation detected is the combination of both signals. However (see Figure 3), the edges of the data and the clock lines are not superposed. Thus, they can be easily separated to obtain independent signals.

Since the falling edges of clock signal will always be at the same place, contrary to the falling edges of data signal, we can use them to improve our trigger model. Indeed we consider the detection of a signal based on 11 equidistant falling edges.

Indirect Emanations. If we compare the data signal and the compromising emanation (see Figure 4) we clearly see that the electromagnetic signal is not directly related to the falling edge, as described by Smulders. Indeed, the durations are not equivalent. Thus, the peaks acquired by our antenna seem to be indirectly generated by the falling edges of the combination of clock and data signals. They are probably generated by a peak of current when the transistor is switched. Nevertheless, these emanations, represented by 14 peaks, 11 for the clock signal and 3 for the data signal (see the horizontal lines in Figure 2 or the peaks in Figure 3) partially describe the logic state of the data signal and can be exploited.

Collisions. Because only the falling edges are detected, eventually collisions occur during the keystroke recovery process. For instance, both E (0x24) and G (0x34)

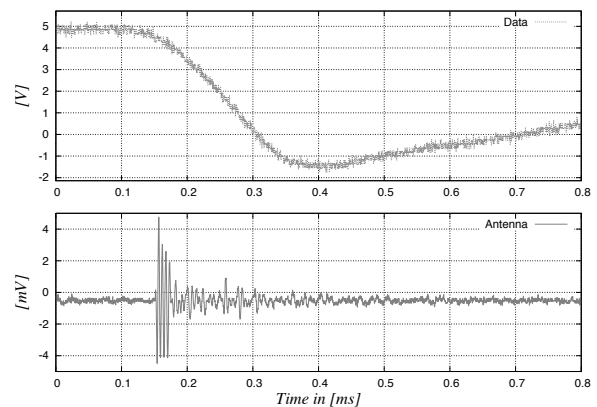


Figure 4: A falling edge of the data signal (upper graph) and the electromagnetic emanation of the keyboard (lower graph). The compromising emission is not directly generated by the data signal such as described by Smulders in [30].

share the same trace if we consider only falling edges. We define the falling edge trace as '2' when both data and clock peaks are detected and '1' when only a clock peak is captured. The letters E (see lower graph in Figure 3) and G may be described by the string 21112112111.

In Figure 5 we grouped every one byte-long scan code, according to their shared falling edge-based traces.

Trace	Possible Keys
21111111111	<non-US-1>
2111111121	<Release key>
21111111211	F11 KP KP0 SL
21111112111	8 u
21111121111	2 a
21111212111	Caps_Lock
21112111111	F4 `
21112112111	- ; KP7
21112121111	5 t
21121111111	F12 F2 F3
2112111121	Alt+SysRq
21121112111	9 Bksp Esc KP6 NL o
21121121111	3 6 e g
21121211111	1 CTRL_L
2112121211	[
21121111111	F5 F7
21121111211	KP- KP2 KP3 KP5 i k
21121112111	b d h j m x
21121121111	SHIFT_L s y
21121121211	' ENTER]
21121211111	F6 F8
21121211211	/ KP4 l
21121212111	f v
21211111111	F9
21211111211	, KP+ KP. KP9
21211112111	7 c n
21211121111	Alt_L w
21211121211	SHIFT_R \
21211211111	F10 Tab
21211211211	. KP1 p
21211212111	Space r
21212111111	F1
21212111211	0 KP8
21212112111	4 y
21212121111	q
21212121211	=

Figure 5: The one byte-long scan codes classification, according to the falling edges trace for an American keyboard layout.

Even if collisions appear, falling edge traces may be used to reduce the subset of possible transmitted scan codes. Indeed, the average number of potential characters for a falling edge trace is 2.4222 (2.0556 if we consider only alpha-numeric characters and a uniform distribution). For example, an attacker who captured the falling edge-based trace of the word password obtains a subset of $3 \cdot 2 \cdot 3 \cdot 3 \cdot 2 \cdot 6 \cdot 2 \cdot 6 = 7776$ potential words, according to Figure 5. Thus, if the objective of the attacker is to recover a secret password, he has significantly reduced the test space (the initial set of $36^8 \approx 2^{41}$ is lowered to 2^{13}). Moreover, if the eavesdropped information concerns an e-mail or a text in English, the plaintext re-

covery process can be improved by selecting only words contained in a dictionary.

Feature Extraction. The recovery procedure is firstly based on a trigger model, able to detect 11 equidistant peaks transmitted in less than 1 ms. Then, we compute the number of peaks, using a peak-detection algorithm and the GNU Radio library. The feature extraction is based on the number of peaks correlated to the most probable value of the table depicted in Figure 5. The main limitation of the recovery procedure is the ability to trigger this kind of signal.

5.2 The Generalized Transition Technique

The previously described attack is limited to a partial recovery of the keystrokes. This is a significant limitation. We know that between two '2' traces, there is exactly one data rising edge. If we are able to detect this transition we can fully recover the keystrokes.

To highlight potential compromising emanations on the data rising edge, we use a software band-pass filter to isolate the frequencies of the broadband impulses (e.g. 105 MHz to 165 MHz of the raw signal in Figure 2). Figure 7 corresponds to the filtered version of the raw time-domain signal represented in Figure 6. We remark

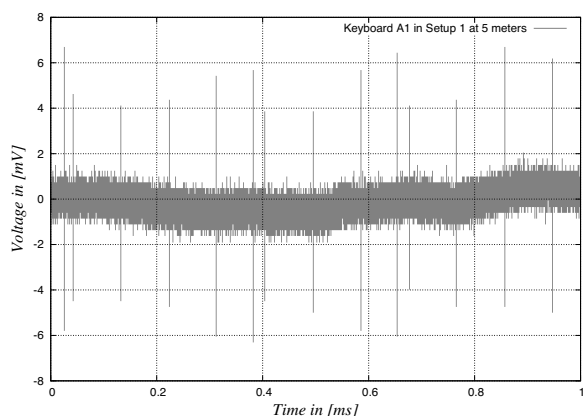


Figure 6: Raw signal (Keyboard A1, Setup 1 at 5 meters with the biconical antenna) when the key E is pressed.

that the filtering process significantly improves the SNR. Thus, the peak detection algorithm is much more efficient.

Furthermore, we notice that the energy of the peaks of the clock falling edges is not constant. Empirically, clock peaks have more energy when the state of data signal is high. Indeed, the data signal pull-up resistor is open. When the clock signal is pulled down, the surplus of energy creates a stronger peak. Hence, the peaks generated by the falling edge of the clock signal intrinsically encode the logic state of the data signal. Because

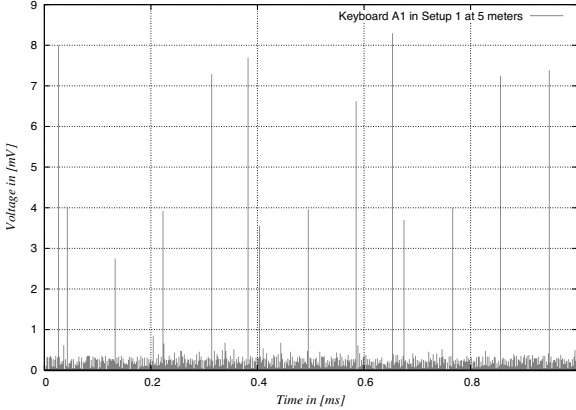


Figure 7: Band-pass (105-165MHz) filtered signal of Figure 6.

there is exactly one rising edge between two falling edge traces of '2', we simply consider the highest clock peak as the rising edge data transition. For example in Figure 7, the rising edge data transitions are respectively at peaks 5 and 9. Thus, the complete data signal is 0010 0100 which corresponds to 0x24 (E). Thus, we manage to completely recover the keystrokes. Note that the band-pass filter improves the previous attack as well. However, the computation cost prevents real time keystroke recovery without hardware accelerated filters.

Feature Extraction. The recovery procedure is firstly based on the same trigger model described previously (11 equidistant peaks detected in less than 1 ms). Then, we filter the signal to consider only the frequency bands containing the peak impulses. The feature extraction is based on the detected peaks. First, we define the threshold between a high peak and a low peak thanks to the two first peaks. Indeed, because we know that data and clock are pulled down, the first one corresponds to a state where clock is high and data is low and the second one describes the state where both signals are low. Then, we determine the potential (and colliding) keystrokes with Figure 5. In our example, it corresponds to the keys 3,6,E,G. Then, we select some bits which differentiate these keys. According to their scan code 3=0x26, 6=0x36, E=0x24, G=0x34 we check the state of the peaks 4 and 8 in Figure 7, which correspond to respectively the second and the fifth bit of the scan codes. Because they are both low, we conclude that the transmitted key is E.

5.3 The Modulation Technique

Figure 2 highlights some carriers with harmonics (vertical lines between 116 MHz and 147 MHz). These compromising electromagnetic emissions come from unin-

tentional emanations such as radiations emitted by the clock, non-linear elements, crosstalk, ground pollution, etc. Determining theoretically the reasons of these compromising radiations is a very complex task. Thus, we can only sketch some probable causes. The source of these harmonics corresponds to a carrier of approximately 4 MHz which is very likely the internal clock of the microcontroller inside the keyboard. Interestingly, if we correlate these harmonics with both clock and data signals, we clearly see modulated signals (in amplitude and frequency) which fully describe the state of both clock and data signals, see Figure 8. This means that the scan code can be completely recovered from these harmonics.

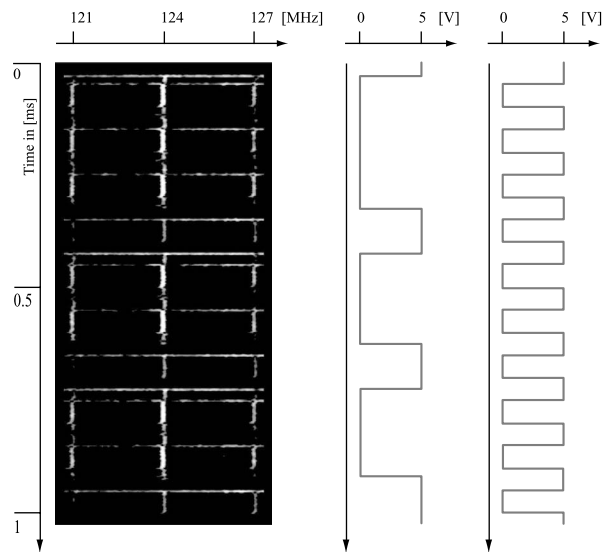


Figure 8: The amplitude and frequency modulations of the harmonic at 124 MHz correlated to both data and clock signals (Keyboard A1, semi-anechoic chamber at 5 meters).

Note that even if some strong electromagnetic interferences emerge, the attacker may choose non-jammed harmonics to obtain a clear signal. It is even possible to superpose them to improve the SNR. Compared to the previous techniques, the carrier-based modulation is much more interesting for distant reception. Indeed, AM and FM transmissions are generally less disrupted by noise and obstacles such as walls, floors, etc. Moreover this technique is able to fully recover the keystrokes. These indirect emanations – which have no formal explanation, but are probably based on crosstalk with the ground, the internal clock of the microcontroller, data and clock signals – let the attacker recover the keystrokes of a keyboard.

This experiment shows that cheap devices such as keyboards may radiate indirect emanations, which are much

more compromising than direct emanations. Even if the SNR is smaller, the use of a frequency modulation significantly improves the eavesdropping range. Moreover, the attacker may avoid some noisy frequency bands by selecting only the clearest harmonics. Furthermore, indirect emanations completely describe both clock and data signals.

Feature Extraction. The feature extraction is based on the demodulation in frequency and amplitude of the captured signal centered on the strongest harmonic. In our example and according to Figure 8 the carrier corresponds to 124 MHz. We used the GNU Radio library to demodulate the signal. However, we still need to use the trigger model based on peak detector since the memory of the oscilloscope is limited. Another option is to directly capture the signal with the USRP. Indeed, the lower but continuous sampling rate of the USRP is sufficient to recover the keystrokes. Unfortunately, the sensitivity of the USRP is weaker than the oscilloscope and the eavesdropping range is limited to less than 2 meters in the semi-anechoic chamber.

5.4 The Matrix Scan Technique

The techniques described above are related to the use of PS/2 and some laptop keyboards. However, new keyboards tend to use USB or wireless communication. In this section, we present another compromising emanation which concerns all keyboard types: PS/2, USB, Notebooks and even wireless keyboards. This attack was previously postulated by Kuhn and Anderson [20] but no practical data has appeared so far in the open literature.

Almost all keyboards share the same pressed key detection routine. A major technical constraint is to consider a key as pressed if the button is pushed for 10 ms, see US Patent [31]. Thus every pressed key should be detected within this time delay. From the manufacturer's point of view, there is another main constraint: the cost of the device. A naive solution to detect pressed keys is to poll each key in a row. This solution is clearly not optimal since it requires a large scan loop routine and thus longer delays. Moreover important leads (i.e. one circuit for each key) increase the cost of the device.

A smart solution [31] is to arrange the keys in a *matrix*. The keyboard controller, often a 8-bit processor, parses columns one-by-one and recovers the state of 8 keys at once. This *matrix scan* process can be described as 192 keys (some keys may not be used, for instance modern keyboards use 104/105 keys) arranged in 24 columns and 8 rows. Columns are connected to a driver chip while rows are connected to a detector chip. Keys are placed at the intersection of columns and rows. Each key is an analog switch between a column and a row. The keyboard controller pulses each column through the driver

(using the address bus and the strobe signal). The detector measures the states of the 8 rows. Note that a row is connected to 24 keys, but only one may be active, the one selected by the driver. Suppose we pressed the key corresponding to column 3 and row 5. The controller pulses columns ..., 22, 23, 24, 1, 2 with no key event. Now, the controller pulses column 3. Row 5, which corresponds to the pressed key, is detected. The keyboard starts a subroutine to transmit the scan code of the key to the computer. This subroutine takes some time. Thus, the next column pulse sent by the scan routine is delayed.

Columns in the matrix are long leads since they connect generally 8 keys. According to [31], these columns are continuously pulsed one-by-one for at least $3\mu s$. Thus, these leads may act as an antenna and generate electromagnetic emanations. If an attacker is able to capture these emanations, he can easily recover the column of the pressed key. Indeed, the following pulse will be delayed.

To figure out if these emanations can be captured, we picked Keyboard A6 and acquired the signal being one meter from the keyboard in the semi-anechoic chamber with a simple one meter long wire of copper as antenna. Figure 9 gives the repeated peak burst continuously emitted by the keyboard. Figure 10 shows the zoomed compromising emanations when the key C resp. key H is pressed.

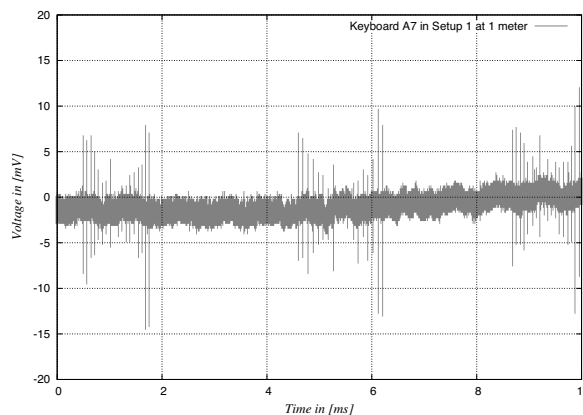


Figure 9: A large view of compromising emanations exploited by the Matrix Scan Technique, (Keyboard A7, semi-anechoic chamber at 1 meter).

The key matrix arrangement may vary, depending on the manufacturer and the keyboard model. We dismantled a keyboard and analyzed the key circuit layout to retrieve the matrix key specifications. The right part of the keyboard layout is depicted on Figure 11. We clearly identify a column (black) and four rows.

Figure 12 represents the groups of alphanumeric scan codes according to their indirect compromising emanations.

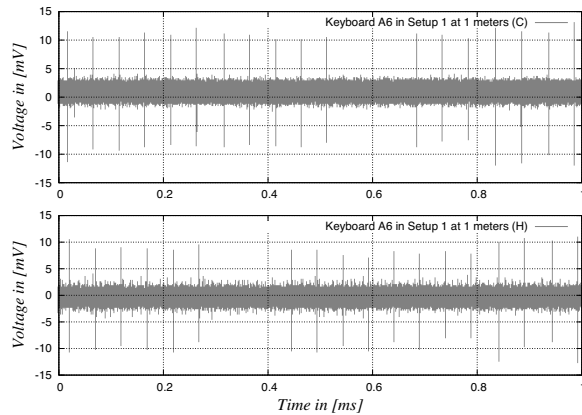


Figure 10: The matrix scan emanations for the letters C and H (Keyboard A6, Setup 1 at 1 meter).

tions (or column number) for Keyboard A6. We describe each electromagnetic signal as a number corresponding to the delayed peak. For example, in Figure 10, the key C is described as 12 and the key H as 7.

Even if this signal does not fully describe the pressed key, it still gives partial information on the transmitted scan code, i.e. the column number. So, as described in the Falling Edge Transition Technique, collisions occurs between key codes. Note that this attack is less efficient than the first one since it has (for this specific keyboard) in average 5.14286 potential key codes for a keystroke (alpha-numeric only). However, an exhaustive search on the subset is still a major improvement.

Note that the matrix scan routine loops continuously. When no key is pressed, we still have a signal composed of multiple equidistant peaks. These emanations may be used to remotely detect the presence of powered computers.

Concerning wireless keyboards, the wireless data burst transmission can be used as an electromagnetic trigger to detect exactly when a key is pressed, while the matrix scan emanations are used to determine the column it belongs to. Moreover the ground between the keyboard and the computer is obviously not shared, thus the compromising electromagnetic emanations are stronger than those emitted by wired keyboards. Note that we do not consider the security of the wireless communication protocol. Some wireless keyboards use a weakly or not encrypted channel to communicate with the computer, see [8, 23].

Feature Extraction. To partially recover keystrokes, we continuously monitor the compromising emanations of the matrix scan routine with a specific trigger model. According to Figure 12 the six first peaks are always present, as well as the last three peaks. Indeed, these peaks are never missing (or delayed). Thus, we use this

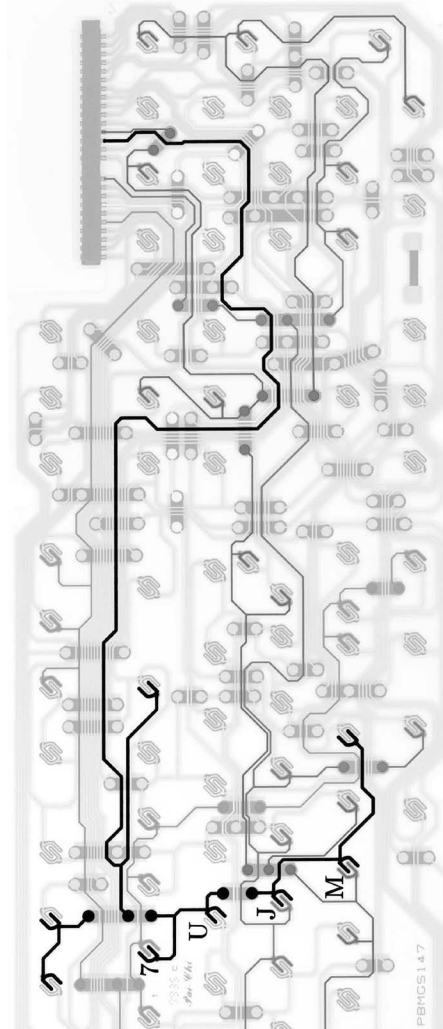


Figure 11: Scan matrix polls columns one-by-one. We are able to deduce on which column the pressed key belongs to. On this keyboard, there will be a collision between keystrokes 7, U, J, M, and others non alpha-numeric keys such as F6, F7, ^, and the dot.

Peak trace	Possible Keys
7	6 7 h J M N U Y
8	4 5 B F G R T V
9	Backspace ENTER
10	9 L O
11	0 P
12	3 8 C D E I K
13	1 2 S W X Z
14	SPACE A Q

Figure 12: The alpha-numeric key classification according to the key scanning routing compromising emanations (Keyboard A6 with American layout).

fixed pattern to define a trigger model. Moreover, the matrix scan continuously radiates compromising emanations since the key is pressed. When a keystroke subset is detected, we acquire multiple samples until another pattern is detected. Therefore, we pick the most often captured pattern.

5.5 Distinguishing Keystrokes from Multiple Keyboards

The falling edge-based traces are distinguishable depending on the keyboard model. Indeed, according to the frequency of the peaks, the clock frequency inconsistencies, the duration between clock and data falling edges, we are able to deduce a specific *fingerprint* for every keyboard. When multiple keyboards are radiating at the same time, we are able to identify and differentiate them. For example, we measured a clock frequency of 12.751 KHz when a key was pressed on a keyboard and the clock frequency was 13.752 KHz when a key was pressed on another keyboard. Thus, when an emanation is captured, we measure the time between two falling edges of the clock and then we deduce if the scan code comes from first or the second keyboard. In practice, we were able to differentiate all the keyboards we tested, even if the brand and the model were equivalent.

This method can be applied to the Falling Edge Transition Technique, the Generalized Transition Technique and the Modulation Technique since they rely on the same kind of signal. The distinguishing process for the Modulation Technique can even be improved by using the clock frequency inconsistencies of the microcontroller as another identifier. For the Matrix Scan Technique, the compromising electromagnetic emanation burst emitted every 2.5 ms (see Figure 9) can be used as a synchronization signal to identify a specific keyboard emission among multiple keyboards. Additionally, the duration between the scan peaks is different, depending on the keyboard model. Thus, it may be used to identify the source keyboard. However, the continuous emission significantly deteriorates the identification process.

Another physical element can be used to distinguish keystrokes from multiple keyboards. For the three first techniques, the broadband impulse range is determined by the length of the keyboard cable, which forms a resonant dipole. Thus, we can use this particularity to identify the source of a compromising emanation. An interesting remark is that the length of the wire connecting the computer to the keyboard is shorter in notebooks. The frequency band of the compromising emanation is higher and the SNR smaller. The Matrix Scan Technique emanates at a higher frequency since the leads of the keyboard layout, acting as an antenna, are shorter.

6 Evaluation in Different Environments

While we have demonstrated techniques that should be able to extract information from keyboard emanations, we have not studied how they are affected by different environments. In this section we study the accuracy of our approaches in all the environments described. Our analysis indicates that keyboard emanations are indeed problematic in practical scenarios.

Evaluating the emission risks of these attacks is not an easy task. Indeed, these results highly depend on the antenna, the trigger model, pass-band filters, peak detection, etc. Moreover, we used trivial filtering processes and basic signal processing techniques. These methods could be significantly improved using beam-forming, smart antennas, better filters and complex triggers. In addition, measurements in real environments but the semi-anechoic chamber were subject to massive change, depending on the electromagnetic interferences. Figure 13 gives the list of vulnerable keyboards in all setups, according to the four techniques previously described. Note that all the tested keyboards (PS/2, USB, wireless and laptop) are vulnerable to at least one of these attacks. First, we present the measurements in Setup 1 (semi-anechoic chamber) to guarantee some stable results.

Keyboard	Type	FETT	GTT	MT	MST
A1	PS/2	✓	✓	✓	✓
A2	PS/2	✓	✓		✓
A3	PS/2	✓	✓	✓	✓
A4	PS/2	✓	✓	✓	
A5	PS/2	✓	✓	✓	
A6	PS/2	✓	✓		✓
A7	PS/2	✓			✓
B1	USB				✓
B2	USB				✓
C1	LT	✓	✓		✓
C2	LT				✓
D1	Wi				✓

Figure 13: The vulnerability of the tested keyboards according to the Falling Edge Transition Technique (FETT), the Generalized Transition Technique (GTT), the Modulation Technique (MT) and the Matrix Scan Technique (MST).

6.1 Results in the Semi-Anechoic Chamber

We consider an attack as successful when we are able to correctly recover more than 95% of more than 500 keystrokes. The Falling Edge Transition Technique, the Generalized Transition Technique and the Modulation Technique are successful in the semi-anechoic chamber

for all vulnerable keyboards. This means that we can recover the keystrokes (fully or partially) to at least 5 meters (the maximum distance inside the semi-anechoic chamber). However, the Matrix Scan Technique is limited to a range of 2 to 5 meters, depending on the keyboard. Figure 14 represents the probability of success of the Matrix Scan Technique according to the distance between the tested keyboard and the antenna.

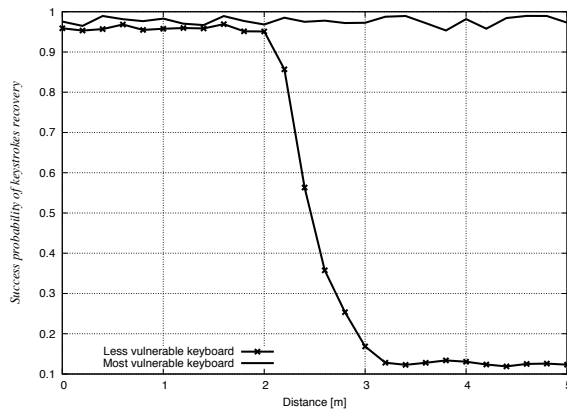


Figure 14: The success probability of the Matrix Scan Technique in the semi-anechoic chamber according to the distance.

We notice that the transition between a successful and a missed attack is fast. Indeed, The correctness of the recovery process is based on the trigger of the oscilloscope. If a peak is not detected, the captured signal is incomplete and the recovered keystroke is wrong. Thus, under a SNR of 6 dB there is nearly no chance to successfully detect the peaks. The SNR is computed according to the average value of the peaks in volts divided by the RMS of the noise in volts.

Considering 6 dB of SNR as a minimum, we are able to estimate the theoretical maximum distance to successfully recover the keystrokes for all techniques in the semi-anechoic chamber. Figure 15 gives the estimated maximum distance range according to the weakest and the strongest keyboard.

In Figure 16 the upper graph gives the SNR of the Falling Edge Transition Technique and the Generalized Transition Technique on Keyboard A1 from 1 meter to 5 meters. The middle graph details the SNR (in dB) of the strongest frequency carrier of the Modulation Technique for the same keyboard. Thus, we can estimate the maximum range of these attacks according to their SNR. The lower graph gives the SNR of the Matrix Scan Technique for the same keyboard. All the measurements were collected in the semi-anechoic chamber.

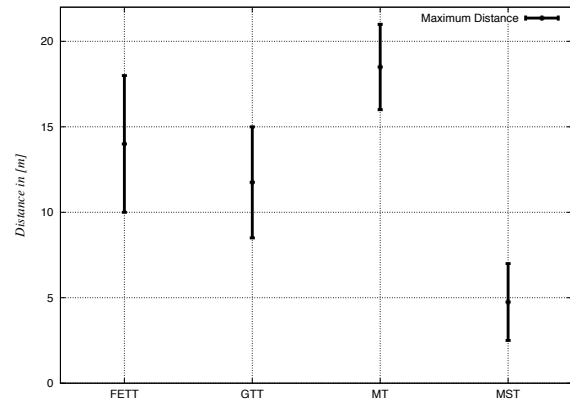


Figure 15: The theoretically estimated maximum distance range to successfully recover 95% of the keystroke according the four techniques in the semi-anechoic chamber, from the less vulnerable to the most vulnerable keyboard.

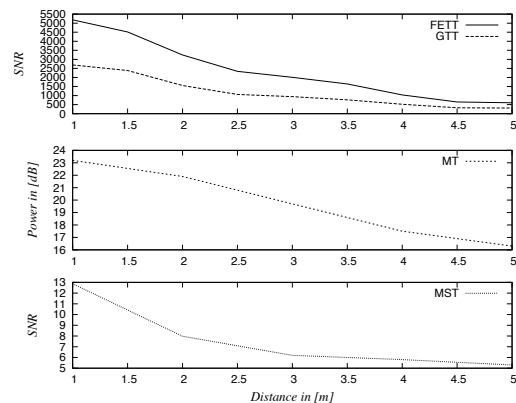


Figure 16: Signal-to-Noise ratio of the peaks [V] / RMS of the noise [V] for the Falling Edge Transition Technique and the Generalized Transition Technique (upper graph). SNR [dB] of the compromising carrier of the Modulation Technique (middle graph). SNR of the peaks [V] / RMS of the noise [V] for Matrix Scan Technique (lower graph).

6.2 Results in Practical Environments

The second phase is to test these techniques in some practical environments. The main difference is the presence of a strong electromagnetic background noise. However, all the techniques remain applicable.

Setup 2: The Office. Figure 17 gives the probability of success of the Generalized Transition Technique on Keyboard A1 measured in the office according to the distance between the antenna and the keyboard. We notice that the sharp transition is present as well when the SNR of the peaks falls under 6 dB. The maximum range of this at-

tack is between 3 and 7.5 meters depending on the tested keyboard. Note that these values were unstable due to a changing background noise. They correspond to an average on multiple measurements.

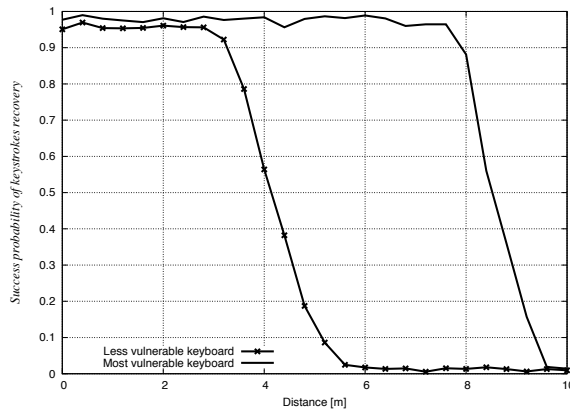


Figure 17: The success probability of the Generalized Transition Technique in the office, according to the distance between the keyboard and the antenna (biconical).

The Modulation Technique is based on a signal carrier. The SNR of this carrier should determine the range of the attack. However, we obtained better results with the same trigger model used in the Falling Edge Transition Technique and the Generalized Transition Technique than one based on the carrier signal only.

Because the Matrix Scan Technique is related to the detection of the peaks, we noticed the same attenuation when the SNR falls under 6 dB. Figure 18 gives the maximum range for the four techniques measured in the office.

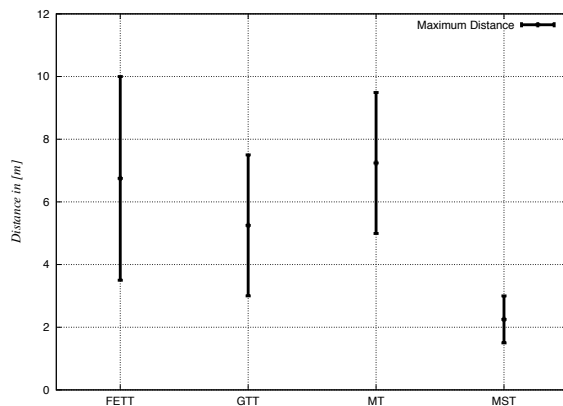


Figure 18: Maximum distance ranges, from the least vulnerable keyboard to the most vulnerable keyboard, to successfully recover 95% of the keystroke according to the techniques (in the office with the biconical antenna).

Setup 3: The Adjacent Office. Results on this setup are basically the same as the previous setup (the office), except that the wall made of plaster and wood removes 3 dB to the SNR.

Setup 4: The Building. We notice some unexpected results in this setup. Indeed, we are able to capture the signal and successfully recover the keystroke with a probability higher than 95% 20 meters away from the keyboard (i.e. the largest distance inside the building). Sometimes the environment can be extremely favorable to the eavesdropping process. For example, metallic structures such as pipes or electric wires may act as antennas and significantly improve the eavesdropping range. In this case, the compromising emanations are carried by the shared ground of the electric line. Thus, the range is defined by the distance between the keyboard and the shared ground and the distance between the shared ground and the antenna. Note that the Matrix Scan Technique is easily disrupted by a noisy shared ground, since the trigger model is more complicated and the emanations weaker. For this technique, we were only able to successfully capture compromising emanations when the keyboard is at less than one meter away from the shared ground. This setup is interesting because it corresponds to a practical scenario where the eavesdropper is placed in the basement of a building and tries to recover the keystrokes of a keyboard at the fifth floor. Unfortunately, it was impossible to provide stable measurements since they highly depend on the environment. We noticed that the main (metallic) water pipe of the building acts as an antenna as well and can be used in place of the shared ground. Furthermore, this *antenna* is less polluted by electronic devices.

Perfect Trigger. We tried the same experiment in the office, but the background noise was too strong. Indeed, we were not able to successfully detect the compromising emissions. However, with a probe physically connected to the data wire, we correctly triggered the emanations. Indeed, the electromagnetic compromising emissions are present in the shared ground. The limitation concerns only the trigger. All the techniques were applicable on the whole floor (about 20 meters) with the keyboard one meter away from the shared ground.

Obviously, you can directly connect the oscilloscope to the shared ground of the building to eavesdrop the keystrokes. Note that an old PC tower used to supply tested keyboards carries the compromising emanations directly through the shared ground. But, this is out of the scope of this paper since we focused our research on electromagnetic emanations only. To avoid such conductive coupling through power supply, we performed our measurements with the keyboards connected to a battery powered laptop.

7 Countermeasures

In this Section, we suggest some possible countermeasures to protect keyboards against the four attacks.

The first solution to avoid the compromising emanations seems trivial. We should shield the keyboard to significantly reduce all electromagnetic radiations. Many elements inside the keyboard may generate emanations: the internal electronic components of the keyboard, the communication cable, and the components of the motherboard inside the computer. Thus, to eliminate these emanations, we have to shield the whole keyboard, the cable, and a part of the motherboard of the computer. We discussed with a manufacturer and he pointed out that the price to shield the entire keyboard will at least double the price of the device. This solution may not be applicable for cost reasons. One can find on the market some keyboards which respect the NATO SDIP-27 standard. All these documents remain classified and no information is available on the actual emission limit or detailed measurement procedures. Another solution is to protect the room where vulnerable keyboards are used. For example, the room can be shielded or a secure physical perimeter can be defined around the room, for instance 100 meters. Attacks 1, 2 and 3 are directly related to the PS/2 protocol. One solution to avoid unintended information leaks is to encrypt the bi-directional serial communication, see [3]. In modern keyboards, one chip contains the controller, the driver, the detector, and the communication interface. So, the encryption may be computed in this chip and no direct compromising emanations related to the serial communication will appear. Attack 4 is related to the scan matrix loop. A solution could be to design a new scanning process algorithm. Even if keyboards still use scan matrix loop routine, there exists some applicable solutions. As described by Anderson and Kuhn [3], the loop routine can be randomized. Actually columns are scanned in the incremental order 1, 2, 3, . . . , 23, 24, but it seems possible to change the order randomly. Moreover, we can add some random delays during the scanning loop process to obfuscate the execution of the subroutine. Both solutions do not avoid electromagnetic emanations, but makes the keystrokes recovery process theoretically impossible. Paavilainen [27] also proposed a solution. It consists in high-frequency filtering matrix signals before they are fed into the keyboard. This will significantly limits compromising electromagnetic emanations.

8 Extensions

Our study has shown that electromagnetic emanations of modern wired and wireless keyboards may be exploited from a distance to passively recover keystrokes. In this

section, we detail some extensions and remarks.

The main limitation of these attacks concerns the trigger of the data acquisition. This can be improved with an independent process, using specific filters between the antenna and the ADC. Additionally, other compromising emanations such as the sound of the pressed key could be used as trigger. Furthermore, modern techniques such as beamforming could significantly improve the noise filtering.

Another improvement would be to simultaneously leverage multiple techniques. For keyboards that are vulnerable to more than one technique, we could correlate the results of the different techniques to reduce uncertainty in our guesses.

Another extension would be to accelerate these attacks with dedicated hardware. Indeed, the acquisition time (i.e. the transfer of the data to a computer), the filtering and decoding processes take time (about two seconds per keystroke). With dedicated system and hardware-based computation such as FPGAs, the acquisition, filtering and decoding processes can obviously be instantaneous (e.g. less than the minimum time between two keystrokes). However, the keystrokes distinguishing process when multiple keyboards are radiating is still difficult to implement especially for the Matrix Scan Technique, since the acquisition process should be continuous.

We spend time experimenting with different types of antennas and analog-to-digital converters. In particular, we used the USRP and the GNU Radio library to avoid the need of an oscilloscope and to obtain a portable version of the Modulation Technique. Indeed, we can hide the USRP with battery and a laptop in a bag, the antenna can be replaced by a simple wire of copper (one meter long) which is taped on the attacker's body hidden under his clothes. With this transportable setup, we are able to recover keystrokes from vulnerable keyboards stealthily. However the eavesdropping range is less than two meters.

9 Conclusion

We have provided evidence that modern keyboards radiate compromising electromagnetic emanations. The four techniques presented in this paper prove that these inexpensive devices are generally not sufficiently protected against compromising emanations. Additionally, we show that these emanations can be captured with relatively inexpensive equipment and keystrokes are recovered not only in the semi-anechoic chamber but in some practical environments as well.

The consequences of these attacks is that compromising electromagnetic emanations of keyboards still represent a security risk. PS/2, USB laptop and wireless

keyboards are vulnerable. Moreover, there is no software patch to avoid these attacks. We have to replace the hardware to obtain safe devices. Due to cost pressure in the design, manufacturers may not systematically protect keyboards. However, some (expensive) secure keyboards already exist but they are mainly bought by military organizations or governments.

The discovery of these attacks was directly related to our method based on the analysis of the entire spectrum and the computation of Short Time Fourier Transform. This technique has some pros such as the human-based visual detection of compromising emanations, the large spectrum bandwidth, the use of the raw signal without RF front-ends and the post-demodulation using software libraries. The cons are the limited memory and the difficulty to obtain efficient triggers. However, for short data bursts, this solution seems relevant.

Future works should consider similar devices, such as keypads used in cash dispensers (ATM), mobile phone keypads, digicodes, printers, wireless routers etc. Another major point is to avoid the use of a peak detection algorithm since it is the main limitation of these attacks. The algorithms of the feature extractions could be improved as well. The correlation of these attacks with non-electromagnetic compromising emanation attacks such as optical, acoustic or time attacks could significantly improve the keystroke recovery process.

We discussed with a few agencies interested by our videos [36]. They confirmed that this kind of attack has been practically done since the 1980's on old computer keyboards, with sharp transitions and high voltages. However, they were not aware on the feasibility of these attacks on modern keyboards. Some of these attacks were not known to them.

Acknowledgments

We gratefully thank Pierre Zweiacker and Farhad Rachidi from the Power Systems Laboratory (EPFL) for the semi-anechoic chamber and their precious advices. We also thank Eric Augé, Lucas Ballard, David Jilli, Markus Kuhn, Eric Olson and the anonymous reviewers for their extremely constructive suggestions and comments.

References

[1] AGRAWAL, D., ARCHAMBEAULT, B., RAO, J. R., AND ROHATGI, P. The EM Side-Channel(s). In *CHES* (2002), B. S. K. Jr., Çetin Kaya Koç, and C. Paar, Eds., vol. 2523 of *Lecture Notes in Computer Science*, Springer, pp. 29–45.

[2] ANDERSON, R. J., AND KUHN, M. G. Soft Tempest – An Opportunity for NATO. *Protecting NATO Information Systems in the 21st Century*, Washington, DC, Oct 25-26 (1999).

[3] ANDERSON, R. J., AND KUHN, M. G. Lost Cost Countermeasures Against Compromising Electromagnetic Computer Emanations. United States Patent US 6,721,324 B1, 2004.

[4] ASONOV, D., AND AGRAWAL, R. Keyboard Acoustic Emanations. In *IEEE Symposium on Security and Privacy* (2004), IEEE Computer Society, pp. 3–11.

[5] BACKES, M., DÜRMUTH, M., AND UNRUH, D. Compromising reflections-or-how to read lcd monitors around the corner. In *IEEE Symposium on Security and Privacy* (2008), P. McDaniel and A. Rubin, Eds., IEEE Computer Society, pp. 158–169.

[6] BALZAROTTI, D., COVA, M., AND VIGNA, G. Clearshot: Eavesdropping on keyboard input from video. In *IEEE Symposium on Security and Privacy* (2008), P. McDaniel and A. Rubin, Eds., IEEE Computer Society, pp. 170–183.

[7] BERGER, Y., WOOL, A., AND YEREDOR, A. Dictionary attacks using keyboard acoustic emanations. In *ACM Conference on Computer and Communications Security* (2006), A. Juels, R. N. Wright, and S. D. C. di Vimercati, Eds., ACM, pp. 245–254.

[8] BRANDT, A. Privacy Watch: Wireless Keyboards that Blab, January 2003. http://www.pcworld.com/article/108712/privacy_watch_wireless_keyboards_that_blab.html.

[9] CHAPWESKE, A. The PS/2 Mouse/Keyboard Protocol. <http://www.computer-engineering.org/>.

[10] CISPR. The International Special Committee on Radio Interference. http://www.iec.ch/zone/emc/emc_cis.htm.

[11] CORRELL, J. T. Igloo White - Air Force Magazine Online 87, 2004.

[12] DYNAMIC SCIENCES INTERNATIONAL, INC. R-1550a tempest receiver, 2008. http://www.dynamicsciences.com/client/show_product/33.

[13] EATSON, J. GNU Octave, 2008. <http://www.gnu.org/software/octave/>.

[14] ETTUS, M. The Universal Software Radio Peripheral or USRP, 2008. <http://www.ettus.com/>.

- [15] FCC. Federal Communications Commission. <http://www.fcc.gov>.
- [16] GANDOLFI, K., MOURTEL, C., AND OLIVIER, F. Electromagnetic analysis: Concrete results. In *CHES* (2001), Çetin Kaya Koç, D. Naccache, and C. Paar, Eds., vol. 2162 of *Lecture Notes in Computer Science*, Springer, pp. 251–261.
- [17] KUHN, M. G. Compromising Emanations: Eavesdropping risks of Computer Displays. *Technical Report UCAM-CL-TR-577* (2003).
- [18] KUHN, M. G. Security limits for compromising emanations. In *CHES* (2005), J. R. Rao and B. Sunar, Eds., vol. 3659 of *Lecture Notes in Computer Science*, Springer, pp. 265–279.
- [19] KUHN, M. G. Dynamic Sciences R-1250 Receiver, 2008. <http://www.cl.cam.ac.uk/mgk25/r1250/>.
- [20] KUHN, M. G., AND ANDERSON, R. J. Soft Tempest: Hidden Data Transmission Using Electromagnetic Emanations. In *Information Hiding* (1998), D. Aucsmith, Ed., vol. 1525 of *Lecture Notes in Computer Science*, Springer, pp. 124–142.
- [21] LOUGHRY, J., AND UMPHRESS, D. A. Information leakage from optical emanations. *ACM Trans. Inf. Syst. Secur.* 5, 3 (2002), 262–289.
- [22] MIL-STD-461. Electromagnetic Interference Characteristics Requirements for Equipment. <https://acc.dau.mil/CommunityBrowser.aspx?id=122817>.
- [23] MOSER, M., AND SCHRODEL, P. 27MHz Wireless Keyboard Analysis Report, 2005. <http://www.blackhat.com/presentations/bh-dc-08/Moser/Whitepaper/bh-dc-08-moser-WP.pdf>.
- [24] MULDER, E. D., ÖRS, S. B., PRENEEL, B., AND VERBAUWHEDE, I. Differential power and electromagnetic attacks on a FPGA implementation of elliptic curve cryptosystems. *Computers & Electrical Engineering* 33, 5-6 (2007), 367–382.
- [25] NALTY, B. C. *The war against trucks: aerial interdiction in southern Laos, 1968-1972*. Air Force History and Museums Program, United States Air Force, 2005.
- [26] NATIONAL SECURITY AGENCY. TEMPEST: A Signal Problem, 2007. http://www.nsa.gov/public_info/_files/cryptologic_spectrum/tempest.pdf.
- [27] PAAVILAINEN, R. Method and device for signal protection. United States Patent US 7,356,626 B2, 2008.
- [28] QUISQUATER, J.-J., AND SAMYDE, D. Electromagnetic analysis (ema): Measures and countermeasures for smart cards. In *E-smart* (2001), I. Attali and T. P. Jensen, Eds., vol. 2140 of *Lecture Notes in Computer Science*, Springer, pp. 200–210.
- [29] SIGBLIPS DSP ENGINEERING. Baudline, 2008. <http://www.baudline.com>.
- [30] SMULDERS, P. The Threat of Information Theft by Reception of Electromagnetic Radiation from RS-232 Cables. *Computers and Security* 9, 1 (1990), 53–58.
- [31] SONDERMAN, E. L., AND DAVIS, W. Z. Scan-controlled keyboard. United States Patent US 4,277,780, 1981.
- [32] SONG, D. X., WAGNER, D., AND TIAN, X. Timing analysis of keystrokes and timing attacks on ssh. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium* (Berkeley, CA, USA, 2001), USENIX Association, pp. 25–25.
- [33] TANAKA, H. Information leakage via electromagnetic emanations and evaluation of tempest countermeasures. In *ICISS* (2007), P. D. McDaniel and S. K. Gupta, Eds., vol. 4812 of *Lecture Notes in Computer Science*, Springer, pp. 167–179.
- [34] VAN ECK, W. Electromagnetic radiation from video display units: an eavesdropping risk? *Comput. Secur.* 4, 4 (1985), 269–286.
- [35] VARIOUS AUTHORS. The GNU Software Radio, 2008. <http://www.gnuradio.org/>.
- [36] VUAGNOUX, M., AND PASINI, S. Videos of the Compromising Electromagnetic Emanations of Wired Keyboards, October 2008. <http://lasecwww.epfl.ch/keyboard/>.
- [37] YOUNG, J. NSA Tempest Documents, 2008. <http://cryptome.info/0001/nsa-tempest.htm>.
- [38] ZHUANG, L., ZHOU, F., AND TYGAR, J. D. Keyboard acoustic emanations revisited. In *ACM Conference on Computer and Communications Security* (2005), V. Atluri, C. Meadows, and A. Juels, Eds., ACM, pp. 373–382.

Peeping Tom in the Neighborhood: Keystroke Eavesdropping on Multi-User Systems

Kehuan Zhang
Indiana University, Bloomington
kehzhang@indiana.edu

XiaoFeng Wang
Indiana University, Bloomington
xw7@indiana.edu

Abstract

A multi-user system usually involves a large amount of information shared among its users. The security implications of such information can never be underestimated. In this paper, we present a new attack that allows a malicious user to eavesdrop on other users' keystrokes using such information. Our attack takes advantage of the stack information of a process disclosed by its virtual file within *procfs*, the process file system supported by Linux. We show that on a multi-core system, the ESP of a process when it is making system calls can be effectively sampled by a "shadow" program that continuously reads the public statistical information of the process. Such a sampling is shown to be reliable even in the presence of multiple users, when the system is under a realistic workload. From the ESP content, a keystroke event can be identified if they trigger system calls. As a result, we can accurately determine inter-keystroke timings and launch a timing attack to infer the characters the victim entered.

We developed techniques for automatically analyzing an application's binary executable to extract the ESP pattern that fingerprints a keystroke event. The occurrences of such a pattern are identified from an ESP trace the shadow program records from the application's runtime to calculate timings. These timings are further analyzed using a Hidden Markov Model and other public information related to the victim on a multi-user system. Our experimental study demonstrates that our attack greatly facilitates password cracking and also works very well on recognizing English words.

1 Introduction

Multi-user operating systems and application software have been in use for decades and are still pervasive today. Those systems allow concurrent access by multiple users so as to facilitate effective sharing of computing

resources. Such an approach, however, is fraught with security risks: without proper protection in place, one's sensitive information can be exposed to unintended parties on the same system. This threat is often dealt with by an access control mechanism that confines each user's activities to her compartment. As an example, programs running in a user's account are typically not allowed to touch the data in another account without the permission of the owner of that account. The problem is that different users do need to interact with each other, and they usually expect this to happen in a convenient way. As a result, most multi-user systems tend to trade security and privacy for functionality, letting certain information go across the boundaries between the compartments. For example, the process status command `ps` displays the information of currently-running processes; while this is necessary for the purpose of system administration and collaborative resource sharing, the command also enables one to peek into others' activities such as the programs they run.

In this paper, we show that such seemingly minor information leaks can have more serious consequences than the system designer thought. We present a new attack in which a malicious user can eavesdrop on others' keystrokes using nothing but her non-privileged account. Our attack takes advantage of the information disclosed by *procfs* [19], the process file system supported by most Unix-like operating systems such as Linux, BSD, Solaris and IBM AIX. *Procfs* contains a hierarchy of virtual files that describe the current kernel state, including statistical information about the memory of processes and some of their register values. These files are used by the programs like `ps` and `top` to collect system information and can also help software debugging. By default, many of the files are readable for all users of a system, which naturally gives rise to the concern whether their contents could disclose sensitive user information. This concern has been confirmed by our study.

The attack we describe in this paper leverages the

procfs information of a process to infer the keystroke inputs it receives. Such information includes the contents of the extended stack pointer (ESP) and extended instruction pointer (EIP) of the process, which are present in the file `/proc/pid/stat` on a Linux system, where `pid` is the ID of the process. In response to keystrokes, an application could make system calls to act on these inputs, which is characterized by a sequence of ESP/EIP values. Such a sequence can be identified through analyzing the binary executables of the application and used as a pattern to fingerprint the program behavior related to keystrokes. To detect the keystroke event at runtime, we can match the pattern to the ESP/EIP values acquired through continuously reading from the `stat` file of the application's process. As we found in our research, this is completely realistic on a multi-core system, where the program logging those register values can run side by side with its target process. As such, we can figure out when a user strokes a key and use inter-keystroke timings to infer the key sequences [26]. This attack can be automated using the techniques for automatic program analysis [20, 23].

Compared with existing side-channel attacks on keystroke inputs [26, 3], our approach significantly lowers the bar for launching a successful attack on a multi-user system. Specifically, attacks using keyboard acoustic emanations [3, 33, 2] require physically implanting a recording device to record the sound when a user's typing, whereas our attack just needs a normal user account for running a non-privileged program. The timing attack on SSH proposed in the prior work [26] estimates inter-keystroke timings from the packets transmitting passwords. However, these packets cannot be deterministically identified from an encrypted connection [13]. In contrast, our attack detects keystrokes from an application's execution, which is much more reliable, and also works when the victim uses the system locally. Actually, we can do more with an application's semantic information recovered from its executable and procfs. For example, once we observe that the same user runs the command `su` multiple times through SSH, we can assume that the key sequences she entered in these interactions actually belong to the same password, and thus accumulate their timing sequences to infer her password, which is more effective than using only a single sequence as the prior work [26] does. As another example, we can even tell when a user is typing her username and when she inputs her password if these two events have different ESP/EIP patterns in an application.

This paper makes the following contributions:

- *Novel techniques for determining inter-keystroke timings.* We propose a suite of new techniques that accurately detects keystrokes and determines inter-keystroke timings on Linux. Our approach includes

an automatic program analyzer that extracts from the binary executable of an application the instructions related to keystroke events, which are used to build a pattern that fingerprints the events. During the execution of the application, we use a shadow program to log a trace of its ESP/EIP values from procfs. The trace is searched for the occurrences of the pattern to identify inter-keystroke timing. Our attack does not need to change the application under surveillance, and works even in the presence of address space layout randomization [29] and realistic workloads. Our research also demonstrates that though other UNIX-like systems (e.g., FreeBSD and OpenSolaris) do not publish these register values, they are subject to similar attacks that utilize other information disclosed by their procfs.

- *Keystroke analysis.* We augmented the existing keystroke analysis technique [26] with semantic information: once multiple timing sequences are found to be associated with the same sequence of keys, our approach can combine them together to infer these keys, which turns out to be very effective. We also took advantage of the information regarding the victim's writing style to learn the English words she types.
- *Implementation and evaluations.* We implemented an automatic attack tool and evaluated it using real applications, including `vim`, `SSH` and `Gedit`. Our experimental study demonstrates that our attack is realistic: inter-keystroke timings can be reliably collected even when the system is under a realistic workload. We also discuss how to defend against this attack.

The attack we propose aims at keystroke eavesdropping. However, the privacy implication of disclosing the ESP/EIP information of other users' process can be much more significant. With our techniques, such information can be conveniently converted to a system-call sequence that describes the behavior of the process, and sometimes, the data it works on and the activities of its users. As a result, sensitive information within the process can be inferred under some circumstances: for example, it is possible to monitor a key-generation program to deduce the secret key it creates for another user, because the key is computed based on random activities within a system, such as mouse moves, keystrokes and networking events, which can be discovered using our techniques.

The information-leak vulnerability exploited by our attack is pervasive in Linux: we checked 8 popular distributions (Red Hat Enterprise, Debian, Ubuntu, Gentoo, Slackware, openSUSE, Mandriva and Knoppix) that represent the mainstream of Linux market [9] and found that all of them publish ESP and EIP. Some other Unix-

like systems, particularly FreeBSD, have different implementations of procfs that do not disclose the contents of those registers to unauthorized party. However, given unrestricted access to procfs, similar attacks that use other information can still happen: for example, we found that `/proc/pid/status` on FreeBSD reveals the accumulated kernel time consumed by the system calls within a process; such data, though less informative than ESP/EIP, could still be utilized to detect keystrokes in some applications, as discussed in Section 6.2. Fundamentally, we believe that the privacy risks of procfs need to be carefully evaluated on multi-core systems, as these systems enable one process to gather information from other processes in real time.

The rest of the paper is organized as follows. Section 2 presents an overview of our attack. Section 3 elaborates our techniques for detecting inter-keystroke timings. Section 4 describes a keystroke analysis using the timings. Section 5 reports our experimental study. Section 6 discusses the limitations of our attack, similar attacks on other UNIX-like systems and potential defense. Section 7 surveys the related prior research, and Section 8 concludes the paper.

2 Overview

This section describes our attack at a high level.

Attack phases. Our attack has two phases: first, the timing information between keystrokes is collected, and then such information is analyzed to infer the related key sequences. These phases and their individual components are illustrated in Figure 1. In the first phase, our approach analyzes the binary executable of an application to extract the ESP/EIP pattern that characterizes its response to a keystroke event, and samples the `stat` file of the application at its runtime to log a trace of those register values. Inter-keystroke timings are determined by matching the pattern to the trace. In the second phase, these timings are fed into an analysis mechanism that uses the Hidden Markov Model (HMM) to infer the characters being typed.

An example. We use the code fragment in Figure 2 as an example to explain the design of the techniques behind our attack. The code fragment is part of an editor program¹ for processing a keystroke input. Upon receiving a key, the program first checks its value: if it is ‘`MOV_CURSOR`’, a set of API calls are triggered to move the cursor; otherwise, the program makes calls to insert the input letter to the text buffer being edited and display its content. These two program behaviors produce two different system call sequences, as illustrated in the figure. This example is written in C for illustration purpose. Our techniques actually work on binary executables.

	System Call Sequence for <code>MOV_CURSOR</code> :	System Call Sequence for insert a char:
1 ...		
2 if (input_ready()) {		
3 c = vgetc();		
4 switch (c) {		
5 ...	read	read
6 MOV_CURSOR: {	select	select
7 ...	select	select
8 cursor_pos_info();	select	select
9 update_cursor();	select	select
10 ...	select	select
11 ...	select	select
12 };	select	select
13 default: { //insert a char	select	select
14 ...	select	write
15 alloc_buf();	write	select
16 insert_char();	select	_llseek
17 update_undo();	select	write
18 flush_buffers;	select	select
19 ...		fsync
20 }		select
21 }		select
22 ...		
23 }		

Figure 2: An Example.

To prepare for an attack, our approach first performs a dynamic analysis on the program’s executable to extract its ESP/EIP pattern that characterizes the program’s response to a keystroke input. Examples of such a response includes allocating a buffer to hold the input (`alloc_buf()`) and inserting it to the text (`insert_char()`). In our research, we found that such a pattern needs to be built upon system calls because sampling of a process’s `stat` file can hardly achieve the frequency necessary for catching the ESP/EIP pairs unrelated to system calls (Section 3.1). When a system call happens, the EIP of the process always points to virtual Dynamic Shared Object (vDSO)² [22], a call entry point set by the kernel, whereas its ESP value reflects the dynamics of the process’s call stack. Therefore, our approach uses the ESP sequence of system calls as the pattern for keystroke recognition. Such a pattern is automatically identified from the executable through a differential analysis or an instruction-level program analysis (Section 3.1).

When the program is running on behalf of the victim, our approach samples its `stat` file to get its ESP/EIP values, from which we remove those unrelated to system calls according to their EIPs. The rest constitutes an ESP trace of the program’s system calls. This trace is searched for the ESP patterns of keystrokes. Note that the trace may only contain part of the patterns: in the example, inserting a character triggers 17 system calls, whereas only 5 - 6 of them appear in the trace. Our approach uses a threshold to determine a match (Section 3.3). Inter-keystroke timings are measured between two successive occurrences of a same pattern.

The timings are analyzed using an *n*-Viterbi algorithm [26] to infer the characters being typed: our approach first constructs an HMM based upon a set of train-

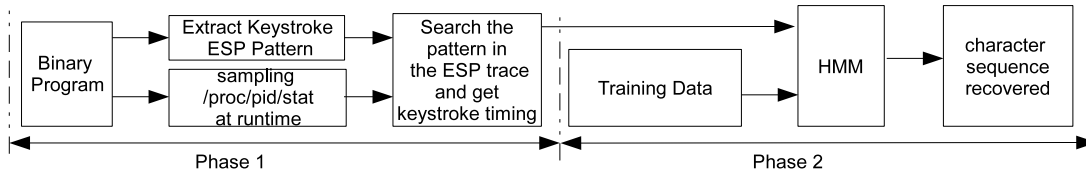


Figure 1: Attack phases

ing data that reflect the timing distributions of different key pairs the victim types, and then runs the algorithm to compute n most likely key sequences with regards to the timing sequence acquired from the ESP trace. We extend the algorithm to take advantage of multiple traces of the same key sequence, which turns out to be particularly effective for password cracking. We also show that the techniques are also effective in inferring English words a user types.

Assumptions. We made the following assumptions in our research:

- *Capability to execute programs.* To launch the attack, the attacker should own or control an account that allows her to execute her programs. This is not a strong assumption, as most users of UNIX-like systems do have such a privilege. The attacker here could be a malicious insider or an intruder who cracks a legitimate user’s account.
- *Multi-core systems.* To detect a keystroke, our shadow process needs to access the ESP of the target process before it accomplishes key-related system calls. However, due to process scheduling, this is not very likely to happen on a single-core system. On one hand, these system calls are typically done within a single time slice. On the other hand, the shadow process often lacks sufficient privileges to preempt the target process when it is working on keystroke inputs, as the latter is usually granted with a high privilege during its interactions with the user. As a result, our process can become completely oblivious to the keystroke events in the target process. This problem is effectively avoided on a multi-core system, which allows us to reliably detect keystroke events in the presence of realistic workloads³, as observed in our experiment (Section 5). Given the pervasiveness of multi-core systems nowadays, we believe that the assumption is reasonable.
- *Access to the victim’s information.* Our attack requires a read access to the victim’s `procfs` files. This assumption is realistic for Linux, on which most part of `procfs` are readable for every user by default. Though one can change her files’ permissions, this can hardly eliminate the problem: all the `procfs` files are dynamically created by the kernel when a new process is forked and their default permissions are

also set by the kernel; as a result, one needs to revise these permissions as soon as she triggers new process, which is unreliable and also affects the use of the tools such as `top`. The fundamental solution is to patch the kernel, which has not been done yet. In addition, we assume that the attacker can obtain some of the text the victim types as training data. This is possible on a multi-user system. For example, some commands typed by a user, such as “`su`” and “`ls`”, causes new processes to be forked and therefore can be observed by other users of the system, which allows the observer to bind the timing sequence of the typing to the content of the text the user entered. As another example, a malicious insider can use the information shared with the victim, such as the emails they exchanged, to acquire the latter’s text and the corresponding timings.

3 Inter-keystroke Timing Identification

In this section, we elaborate our techniques for obtaining inter-keystroke timings from a process.

3.1 Pattern Extraction

The success of our attack hinges on accurate identification of keystroke events from the victim’s process. We fingerprint such an event with an ESP pattern of the system calls related to a keystroke. The focus on system calls here comes from the constraints on the information obtainable from a process: on one hand, a significant portion of the process’s execution time can be spent on system calls, particularly when I/O operations are involved; on the other hand, our approach collects the process’s information through system calls and therefore cannot achieve a very high sampling rate. As a result, the shadow program that logs ESP/EIP traces is much more likely to pick up system calls than other instructions. In our research, we found that more than 90% of the ESP/EIP values collected from a process actually belong to system calls. Note that a process’s EIP when it is making a system call always points to `vDSO`. It is used in our research to locate the corresponding ESP whose content is much more dynamic and thus more useful for fingerprinting a keystroke event.

Our approach extracts the ESP pattern through an automatic analysis of binary executables. This analysis is conducted offline and in an environment over which the attacker has full control. Following we present two analysis techniques, one for the programs that execute in a deterministic manner and the other for those whose executions are affected by some random factors.

Differential analysis. Many text-based applications such as `vim` are deterministic in the sense that two independent runs of these applications under the same keystroke inputs yield identical system call traces and ESP sequences. The ESP patterns of these applications can be easily identified through a differential analysis that compares the system call traces involving keystroke events with those not. Specifically, our program analyzer uses `strace` [27] to intercept the system calls of an application and record their ESP values when it is running. An ESP sequence is recorded before a keystroke is typed, and another sequence is generated after the keystroke occurs⁴. The ESP pattern for a keystroke event is extracted from the second sequence after removing all the system calls that happen prior to the keystroke, as indicated by the first sequence. To ensure that the pattern does not contain any randomness, we can compare the ESP trace of typing the same character twice with the one involving only a single keystroke to check whether the ESPs associated with the second keystroke are identical to those of the first one. The same technique is also applied to test different keys that may have discrepant patterns. In the example described in Figure 2, the ESP sequence of `vim` before Line 2 is dropped from the traces involving keystrokes and as a result, the system calls triggered by the instructions from Line 7 to 11 are picked out as the fingerprint for ‘`MOV_CURSOR`’ and those between Line 14 and 19 identified as the pattern for inserting a letter.

The ESP pattern identified above will go through a false positive check to evaluate its accuracy for keystroke detection. In other words, we want to know whether the pattern or a significant portion of it can also be observed when the user is not typing. This is achieved in our research through searching for the pattern in an application’s ESP trace unrelated to keystroke inputs. Specifically, our analyzer logs the execution time between the first and the last system calls on the pattern, and uses this time interval to define a duration window on the trace, which we call *trace window*. The trace window is slid on the trace to determine a segment against which the pattern is compared. For this purpose, every ESP value on the trace is labeled with the time when its corresponding system call is invoked. The trace window is first located prior to the first ESP value on the trace. Then, it is slid rightwards: each slide either moves an ESP into the window or moves one outside the window. After a slide, our analyzer attempts to find the longest com-

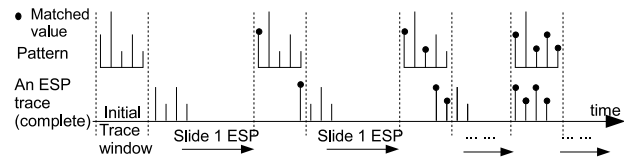


Figure 3: A false positive check. Spikes in the figure represent ESP values.

mon sequence between the trace segment within the window and the pattern. This is the well-known LCS problem [4], which can be efficiently solved through dynamic programming [15]. The size of such a sequence, which we call an *FP level*, is recorded. As such, our approach keeps on sliding the trace window to measure FP levels until all the ESP values on the trace have left the window.

Figure 3 presents an example that shows how the algorithm works. In the initial state, the trace window is located before the first ESP value. Then the trace window starts to slide right to include the first ESP value, which gives a FP level of one. After the window slides again to include one more ESP value, our algorithm returns a common sequence with two members. This process continues, and finally, the window is moved to embrace all four trace members and we observe an FP level of four. This algorithm identifies the portion of the pattern that can show up in absence of keystrokes. The size of the portion, as indicated by the FP level, is used to determine a threshold for recognizing keystrokes from an incomplete ESP trace sampled from a process, which is elaborated in Section 3.3.

Instruction-level analysis. Applications with graphic user interfaces (GUI) can work in a non-deterministic manner: these applications are event-driven and can change their system-call behaviors in response to the events from operating systems (OS), which can be unpredictable. For example, `Gedit` uses a timer to determine when to flash its cursor; the timer, however, can be delayed when the process is switched out of the CPU, which causes system call sequences to vary in different runs of the application. To extract a pattern from these applications, we adopted an instruction-level analysis as described below.

Under Linux, many X-Window based applications are developed using the GIMP Toolkit (aka. `GTK+`) [28]. `GTK+` uses a standard procedure to handle the keystroke event: a program uses a function such as `gtk_main_do_event(event)` to process event; when a key is pressed⁵, this function is invoked to trigger a call-back function of the keystroke event. In our research, we implemented a `Pin` [20] based analysis tool that automatically analyzes a binary executable at the instruction level to identify such a function. After a key has been typed, our analyzer detects the keystroke

event from the function's parameter and from that point on, records all the system calls and their ESPs until the executable is found to receive or dispatch a new event, as indicated by the calls to the functions like `g_main_context_acquire()`. All these system calls are thought to be part of the call-back function and therefore related to the keystroke event⁶. The pattern for keystroke recognition is built upon these calls. We also check false positives of the pattern, as described before.

3.2 Trace Logging

Our attack eavesdrops on the victim's keystrokes through shadowing the process that receives her keystroke inputs. Our shadow process stealthily monitors the target process's keystroke events by keeping track of its ESP/EIP values disclosed by its `stat` file. Since the attack happens in the userland, the attacker has to use system calls to open and read the file. Moreover, a more efficient approach, memory mapping through `mmap()`, does not work on the virtual file that exists only in memory. These issues prevent the shadow process from achieving a high sampling rate. For example, a program we implemented for evaluating our approach updated ESP/EIP values every 5 to 10 microseconds. As a result, we could end up with an incomplete ESP/EIP trace of the target process. This, however, is sufficient for determining inter-keystroke timings, as we found in our research (Section 3.3).

Trace logging with full steam can cost a lot of CPU time. If the activity drags on, suspicions can be roused and alarms can be triggered. To avoid being detected, our attack takes advantage of the semantic information recovered from procs and the target application to concentrate the efforts of data collection on the time interval when the victim is typing the information of interest to the attacker. For example, the shadow process starts monitoring the victim's SSH process at a low rate, say once per 100 milliseconds; once the process is observed to fork a `su` process, our shadow process immediately increases its sampling rate to acquire the timings for the password key sequence. Another approach is using an existing technique [32] to hide CPU usage: UNIX-like systems keep track of a process's use of CPU according to the number of ticks it consumes at the end of each tick; the trick proposed in [32] lets the attack process sleep just before the end of each tick it uses and as a result, OS will schedule a victim process to run and bill the whole tick to that victim process instead of the attack process. We implemented this technique and found that it was very effective (Section 5).

3.3 Timing Detection

We determine inter-keystroke timings from the time intervals between the occurrences of a pattern on an ESP trace sampled from an application's system calls. Two issues here, however, complicate the task. First, some Linux versions may run the mechanisms for address space layout randomization (ASLR) [29] that can cause the ESP values on the pattern to differ from those on the trace. Second, the trace can be incomplete, containing only part of the system calls on the pattern, which makes recognition of the pattern nontrivial. Following we show how these issues were handled in our research.

ASLR performed by the tools such as `Pax` [30] involves randomly arranging the locations of an executable's memory objects such as stack, executable image, library images and heap. It is aimed at thwarting the attacks like control-flow hijacking that heavily rely on an accurate prediction of target memory addresses. Though the defense works on the attacks launched remotely, it is much less effective on our attack, which is commenced locally. Specifically, the address for the bottom of a process's stack can be found in its `stat` and `/proc/PID/maps`⁷. This allows us to "normalize" the ESP values on both the trace and the pattern with the differences between the tops of the stack, as pointed by the ESPs, and their individual bottoms. Neither does ASLR prevent us from correlating an ESP/EIP pair on a trace to a system call, though the knowledge about the vDSO address may not be publically available on some Linux versions: we can filter out the pairs unrelated to system calls according to the observation that the vast majority of the members on the trace actually belong to system calls and therefore have the same EIP values.

To recognize an ESP pattern from an incomplete ESP trace of system calls, we use a threshold τ : a segment of the trace, as determined by the trace window, is deemed matching the pattern if it contains at least τ ESP values of system calls and the sequence of these values also appear on the pattern. The threshold here can be determined using the results of the false positive test described in Section 3.1. Let h be the highest FP level found in the test, and s be the number of the system calls that our shadow process can find from a process when a keystroke occurs. We let $\tau = h + 1$ if $s > h$. Intuitively, this means that a trace segment is considered matching the pattern if it does not contain any ESP sequences not on the pattern and no segments unrelated to keystrokes can match as many ESP values on the pattern as that segment does⁸. If $s \leq h$, we have to set $\tau = s$ because we cannot get more than s ESP samples for every keystroke when monitoring a process. Several measures can be taken to mitigate the false positives that threshold could bring in. One approach is to leverage the observation that people typ-

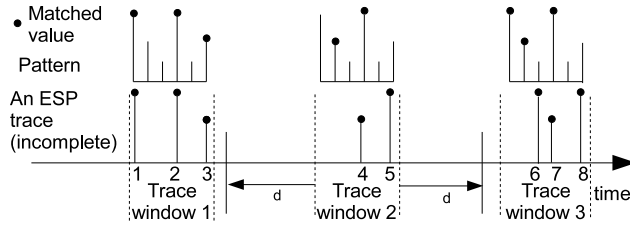


Figure 4: Using time frame d to remove possible false positive matches

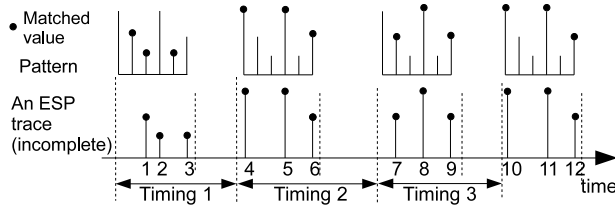


Figure 5: Pattern matching on an ESP trace and the timing interval

ically type more than one key within a short period of time. Therefore, we can require that a segment matching a pattern according to τ be preceded or followed by another pattern-matching segment within a predetermined time frame d , before both of them can be deemed to be indicative of keystroke events. Figure 4 presents an example in which the segment within the Window 2 is not treated as a match to the pattern because there is no other matches happening within the time frame d either before or after the window. In another approach, we use the execution time of a process to estimate the time point when it starts receiving keystrokes, which helps avoid searching the trace unlinked to keystrokes.

After normalizing ESP values and determining the threshold τ , our approach starts searching the trace sampled from the victim's process for the occurrences of the pattern. The searching algorithm we adopted slides the trace window in the same way as the false positive check does (Section 3.1). For each slide, an LCS problem is solved to find the longest common sequence between the trace segment in the window and the pattern. If the length of the sequence is no less than τ and every member on the segment is also on the sequence, the segment is labeled as a match. Once a match is found, we slide the window rightwards to pass all trace members within a short time interval that describes the minimal delay between two consecutive keystrokes, and then start the next round of searching. This process continues until all trace members pass the window. Then, our approach determines timings from the segments labeled as matches: the time interval between two such segments is identified as an inter-keystroke timing if there is no other labeled segments in-between and the duration of the interval is below a predetermined threshold that serves to rule out the

long latencies caused by intermittent typing. An example for illustrating the algorithm is presented in Figure 5, in which the trace window locates four matches with $\tau = 3$, and the durations between these matches are picked out as inter-keystroke timings.

4 Keystroke Analysis

In this section, we describe how to use inter-keystroke timings to infer the victim's key sequence. Our approach is built upon the technique used in the existing timing attack [26]. However, we demonstrate that the technique can become much more effective with the information available on a multi-user system.

4.1 HMM-based Inference of Key Sequences

A Hidden Markov Model [24] describes a finite stochastic process whose individual states cannot be directly observed. Instead, the outputs of these states are visible and therefore can be used to infer the existence of these states. An HMM, like a regular Markov model, assumes that the next states a system can move into only depend on the current state. In addition, it has a property that the outputs of a state are completely determined by that state. These two properties allow a hidden sequence to be easily computed and therefore make the model a pervasive tool for the purposes such as speech recognition and text modeling.

Prior research [26] models the problem of key inference using an HMM. Specifically, let K_0, \dots, K_T be the key sequence typed by the victim, and $q_t \in Q$ ($1 \leq t \leq T$) be a sequence of states representing the key pair (K_{t-1}, K_t) , where Q is the set of all possible states. In each state q_t , an inter-keystroke latency y_t with a Gaussian-like distribution can be observed. Our objective is to find out the hidden states (q_1, \dots, q_T) from the timings (y_1, \dots, y_T) . This modeling is simple and was shown to work well in practice [26], and is further confirmed by our research, though it has oversimplified the relations between the characters being typed: particularly, the chance for a letter to appear at a certain position in an English word may actually relate to all other letters before it, which invalidates the HMM assumption that a transition from q_t to q_{t+1} depends only on q_t .

The HMM for key inference can be solved using the Viterbi algorithm [24], a dynamic programming algorithm that computes the most likely state sequence (q_1, \dots, q_T) from the observed timing sequence (y_1, \dots, y_T) . Let $V(q_t)$ be the probability of the sequence that most likely ends in q_t at time t . The algorithm computes $V(q_t)$ through two steps. In the first step, we assign a set of initial probabilities $V(q_1) =$

$Pr[q_1|y_1]$. The second step inductively computes $V(q_t)$ for every $1 < t \leq T$ and every $q_t \in Q$ as $V(q_t) = \max_{q_{t-1}} Pr[y_t|q_t]Pr[q_t|q_{t-1}]V(q_{t-1})$, where $Pr[y_t|q_t]$ can be estimated from a set of training data (the third assumption in Section 2) and $Pr[q_t|q_{t-1}]$, the transition probability, comes from a uniform distribution over the states reachable from q_{t-1} . This step also keeps track of all the prior states on the sequence with the probability $V(q_t)$. The most likely sequence is identified from the state q_T that maximizes $V(q_T)$. A direct application of this approach, however, does not work well in practice, because even the most likely sequence usually has a very small probability to match the real keystroke inputs. This problem is mitigated in the prior work [26] that extends the algorithm to the n -Viterbi algorithm so as to return the top n most likely sequences given a timing sequence. The difference here is that the n -Viterbi algorithm changes the inductive step (the second step) to identify the sequences with the n largest probabilities. The details of the algorithm can be found in [26].

4.2 Password Cracking

The effectiveness of the n -Viterbi algorithm can be significantly improved with the information available on a multi-user system. Particularly, the name of a process and its owner can be directly found from procs or indirectly from running commands such as ps or top. Once the same user is observed to run the same application multiple times and if such interactions happen within a no-so-long period of time and all involve typing passwords, a reasonable assumption we can make is that all these passwords are actually the same. Therefore, we can combine together the timing sequences recorded from individual interactions to infer a key sequence. Following we describe two ways to do that.

Our first approach is simply averaging all the timings for every key pair to create a new sequence and run the n -Viterbi algorithm over it. Formally, given m timing sequences $(y_1^1, \dots, y_T^1), \dots, (y_1^m, \dots, y_T^m)$, we can compute a new sequence (y_1, \dots, y_T) , where $y_t = \frac{1}{m} \sum_{1 \leq i \leq m} y_t^i$ and $1 \leq t \leq T$. The rationale here is that the distribution of the timing y_t^i of a key pair q_t is a Gaussian-like unimodal distribution and therefore the probability $Pr[y_t|q_t]$ in the inductive step of the algorithm is maximized when y_t becomes the mean of the distribution, which is approximated by averaging all y_t^i . This approach works particularly well when the means of two key pairs are not extremely close.

The other approach, which we call the m - n -Viterbi algorithm, utilizes multiple observations to perform the inductive step of the original algorithm. Specifically, our approach replaces $Pr[y_t|q_t]$ in that step with $Pr[y_t^1, \dots, y_t^m|q_t] = Pr[y_t^1|q_t] \dots Pr[y_t^m|q_t]$ given

these observations (y_t^1, \dots, y_t^m) are independent from each other. This treatment works even in the presence of the key pairs with very close timing distributions. However, it needs a large number of timing sequences to get a good outcome.

Our research shows that both approaches can significantly shrink the space for searching a password. Actually, in our experiment (Section 5.2), we found that using 50 timing sequences, our techniques sped up the password searching by factors ranging from 250 to 2000.

4.3 English Text

Recovery of English text from a timing sequence is no less challenging than password cracking. A password can be figured out through testing many candidates against the target application or a hashed password list. However, the same trick cannot be played on English words because no application and password list can tell you whether you made a right guess. All that we can do is to check all the combinations of the possible words to see whether a meaningful sentence comes out, which becomes a daunting task if the list of such words is long. Moreover, it can be more difficult to find multiple timing sequences associated with the same text, and therefore the aforementioned approaches become less applicable. On the bright side, English words are much less random than passwords: the letters they include and the combinations of those letters have distributions with low entropies. Such a property can be leveraged to adjust the transition probabilities of an HMM to improve the outcomes of key sequence inference. Here we elaborate such techniques used in our research.

A prominent property of English text is use of the SPACE character to separate words. People tend to type the letters in a word faster than SPACE, a signal for a transition between words. This gives the character an identifiable timing feature: typically the key pair involving SPACE incurs longer inter-keystroke latency than other pairs, as illustrated in Figure 6. In our research, we detected SPACE by checking if the timing interval is larger than a predetermined threshold. This threshold can be determined from the training data collected from the victim's typing. Knowledge about the SPACE key helps us to divide a long timing sequence into a collection of small sequences, with each of them representing a word, and then learn these words one by one.

Another important property of English text is its distinct distribution of letters. It is well known that some letters such as 'e' occur more frequently than others, and some bigrams like 'th' and trigrams like 'ion' are also pervasive in a meaningful text. This fact has been leveraged by frequency analysis to crack classic ciphers [1]. The same game can also be played to make key se-

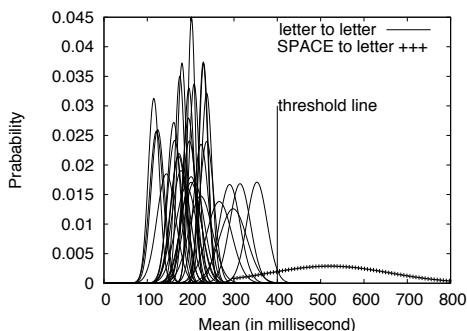


Figure 6: Timing Distribution of SPACE-letter pair, letter-letter pair and threshold

quence inference more effective: we can adjust the transition probabilities of an HMM to ensure that the transition between certain states such as ('i', 'o') to ('o', 'n') is more likely to happen than others. These probabilities can be conveniently obtained from various public sources [18, 10] that provide the statistics of common English text. Such statistics can be further tuned to the victim's writing style according to public writing samples such as her web pages and publications. Moreover, it comes with no surprise that users on the same system are often related: for example, they could all belong to one organization. This allows the attacker to get familiar with the victim's writing from the information they exchanged, for example, the emails between them. In addition, since the timing sequence corresponding to such information can also be identified using our technique, the attacker can actually use the information as the training data for estimating the timing distributions of different key pairs the victim typed.

5 Evaluation

In this section, we describe an experimental study of the attack techniques we propose. Our objective is to understand whether these techniques present a realistic threat. To this end, we evaluated them using 3 common Linux applications: `vim`, `SSH` and `Gedit`. In our experiments, we first ran our approach to automatically extract timing sequences when a user was typing, evaluated the accuracy of these timings and the effectiveness of the attack under different workloads. Then, we analyzed them using our techniques to study how much keystroke information could be deduced. Our experiments were mainly carried out on a computer with a 2.40GHz Core 2 Duo processor and 3GB memory, on which we conducted our study under three Linux versions: RedHat Enterprise Linux 4.0, Debian 4.0 and Ubuntu 8.04. We found that our techniques worked effectively even in the presence of realistic workloads on the server. This suggests that

Table 1: Normalized ESP pattern values (include system calls)

vim		ssh		gedit	
SysCall	ESP	SysCall	ESP	SysCall	ESP
read	1628	rt_sigprocmask	4932	gettimeofday	3624
select	1604	rt_sigprocmask	4932		
select	1876	read	20908		
select	2244	select	4548		
select	1540	rt_sigprocmask	4932		
select	1908	rt_sigprocmask	4932		
select	1556	write	37436		
select	1924	ioctl	37500		
select	1604	select	4548		
write	1548	rt_sigprocmask	4932		
select	1972	rt_sigprocmask	4932		
_llseek	1876	read	37436		
write	1836	select	4548		
select	2180	rt_sigprocmask	4932		
fsync	1752	rt_sigprocmask	4932		
select	2148	write	4620		
select	1972	select	4548		

the information leaks caused by procs can be a real security problem.

5.1 Inter-keystroke Timings

As the first step of our evaluation, we applied our technique to identify the timings from `vim`, `SSH` and `Gedit` on a multi-core system.

vim. `vim` is an extremely common text editor, which is supported by almost all Linux versions. It fits well with the notion of deterministic programs as discussed in Section 3.1, because independent runs of the application with the same inputs always produce the same system call sequence and related ESP sequence. This property enabled us to identify its ESP pattern for a keystroke event using the differential analysis. The pattern we discovered for inserting a letter includes 17 calls. These calls and their normalized ESP values are presented in Table 1. We further ran the application from a user account to enter words, and in the meantime, launched a shadow process from another account to collect the ESP trace of the application. From the trace, our approach automatically identified all the keystrokes we typed. Table 2 shows a trace segment corresponding to two keystrokes, which involves 5 system calls for each keystroke.

In order to evaluate the accuracy of the timing sequence our shadow process found, an instrumented version of `vim` was used in our experiment, which recorded the time when it received a key from `vgetc()`. Such information was used to compute a real timing sequence. We compared these two sequences and found that the de-

Table 2: Examples of ESP traces (values that appear in the pattern are in bold font).

vim	ssh	gedit
1604	4548	520
2244	4932	2988
1908	20908	3052
1924	4548	696
1972	37500	3624
1604	4548	3068
2244	37436	2988
1908	4932	696
1924	4620	520
1972	4548	2988

viations between corresponding timings were at most 1 millisecond, below 3% of the average standard deviation of the timings of different key pairs, as illustrated in Table 3. This demonstrates that the timings extracted from the process were accurate.

SSH. The Secure Shell (SSH) has long been known to have a weakness in its interactive mode, where every keystroke is transmitted through a separate packet and immediately after the key is pressed. This weakness can be exploited to determine inter-keystroke timings for inferring the sensitive information a user types, such as the password for `su`. Prior work [26] proposes an attack that eavesdrops on an SSH channel to identify such timings. A problem of the attack, as pointed out by SSH Communications Security, is that determination of where a password starts in an encrypted connection can be hard [25]. This problem, however, does not present a hurdle to our attack, because we can easily find out from `procs` when `su` is spawned from an SSH process, and start collecting information from SSH from then on. This is exactly what we did in our experiment.

Using the differential analysis, our approach automatically discovered an ESP pattern from SSH when a key was typed for entering a password for `su`. We further ran a shadow process to monitor another user’s SSH process: as soon as it forked an `su` process, our shadow process started collecting ESP values from the SSH process’s `stat` file. The trace collected thereby was compared with the pattern to pinpoint keystroke events and gather the timings between them. The pattern that we found in our experiment included 17 system calls, of which 7 to 10 appeared in every occurrence of the pattern on the trace. The detailed experimental results are in Table 1 and Table 2.

Verification of the correctness of those timings turned out to be more difficult than we expected. `su` does not read password characters one by one from the input. Instead, it takes all of them after a RETURN key has been stroked. Therefore, instrumentation of its source code

Table 3: Examples of the timings measured from ESP traces (Measured) and the real timings (Real) in milliseconds.

Timings	vim		ssh		Gedit	
	measured	real	measured	real	measured	real
1	80	81	135	135	301	303
2	139	139	124	123	285	285
3	88	88	103	103	259	259
4	101	101	110	109	236	236
5	334	335	134	134	181	182
6	86	87	111	110	265	265
7	124	124	132	132	174	174

will not give us the real timing sequence. We solved this problem by replacing `su` with another program that recorded the time when it received a key from SSH, and used such information to generate a timing sequence. This sequence was found to be very close to the one we got from the trace collected by our shadow process, as described in Table 3. We further employed the timings obtained from `su` to infer the passwords being typed, which we found to be very effective (Section 5.2).

Gedit. Gedit is a text editor designed for the X Window system. Like many other applications based upon the GTK+, it is non-deterministic in the sense that two independent runs of the application under the same inputs often produce different system call sequences. In our experiment, we performed an instruction-level analysis of its binary executables using the Pin-based tool we developed. This analysis revealed the call-back function of the key-press event, from which we extracted the system call sequence and related ESP sequence. An interesting observation is that Gedit actually does not immediately display a character a user types: instead, it put the character to a buffer through a GTK+ function `gtk_text_buffer_insert_interactive_at_cursor()`, which does not involve any system calls, and the content of the buffer is displayed when it becomes full or a timer expires. As a result, we could not count on the system calls involved in such a display process for fingerprinting keystrokes. Actually, only one system call was found to be present every time when a key was received: `gettimeofday()`, a call that Gedit uses to determine when to auto-save the document the user is editing. This call seems too general. However, its ESP value turned out to be specific enough for a pattern: in our false positive check, we did not find any other system calls within the application that also had the same ESP. Moreover, our shadow process always caught that ESP whenever we typed. Therefore, this ESP value was adopted as the pattern in our experiment. We further instrumented Gedit to dump the time when this call was invoked for calculating the real timing sequence. Table 1 shows that this sequence is very close to the one collected by our shadow process.

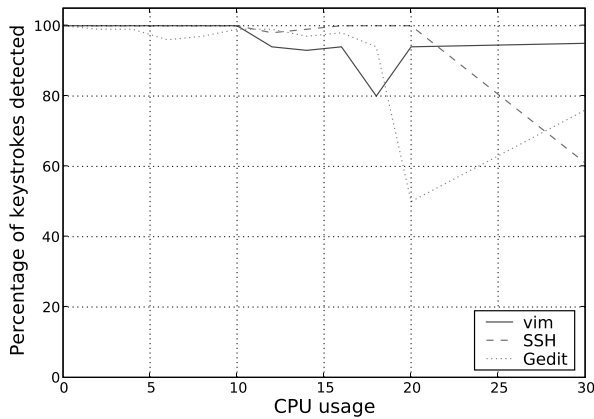


Figure 7: Percentage of keystrokes detected vs. CPU usage

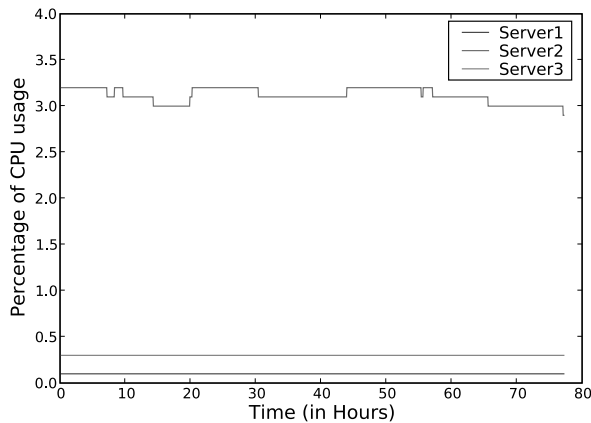


Figure 8: CPU usages of three real-world servers during 72 hours

Impacts of server workloads. A multi-user system often concurrently serves many users. These users' activities could interfere with the collection of inter-keystroke timings. This problem was studied in our research through evaluating the effectiveness of our attack under different workloads. Specifically, we ran our attacks on `vim`, `SSH` and `Gedit` under different CPU usages to measure the percentage of the keystrokes still detectable to our shadow process. The experimental results are elaborated in Figure 7. Here, we sketch our findings.

We found that the impacts of workloads varied among applications. The attacks on `vim` and `SSH` appear to be quite resilient to the interferences from other processes: our shadow process picked up 100% keystrokes for both applications when CPU usage was no more than 10% and still detected 94% from `vim` when the usage went above 20%. In contrast, the attack on `Gedit` was less robust: we started missing keystrokes when more than 2% of

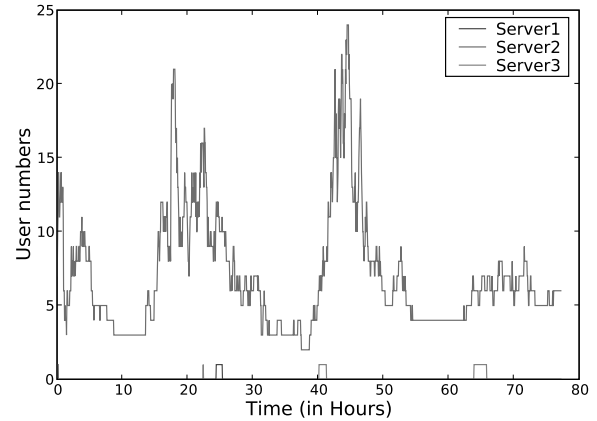


Figure 9: Variations of user numbers on the three servers during 72 hours

CPU time was consumed by other processes. This discrepancy comes from applications' ESP patterns: those involving more system calls are easier to detect.

On the other hand, the workloads on a real-world system are reasonable enough to be handled by our attack. Figure 8 and 9 reports the CPU usages and user numbers we measured from three real-world systems, including a Linux workstation in a public machine room (Server 1), a server for students' course projects (Server 3) and a web server of Indiana University that allows SSH connections from its users (Server 2). The number of users on these systems range from 1 to 24. Our 72-hour monitoring reveals that for 90 percent of time, the CPU usages of these servers were below 3.2%.

We also implemented the technique proposed in [32] to hide the CPU usage of our shadow process. As a result, the process appeared to consume 0% of CPU, as observed from `top`. The cost, however, was that it only reliably identified about 50% of keystrokes we entered. Nevertheless, this still helped inference of keys, particularly when the same input from a user (e.g., password) was sampled repeatedly, as discussed in Section 4.2.

5.2 Key Sequence Inference

We further studied how to use the timings to infer key sequences. Experiments were conducted in our research to evaluate our techniques using both passwords and English words. Here we report the results.

Password. To study the effectiveness of our approach on passwords, we first implemented the n -Viterbi algorithm [26] and used it to compute a baseline result, and then compared the baseline with what can be achieved by the analysis using multiple timing sequences, as described in Section 4.2. Our experiment was carefully

Table 4: The percentage of the search space the attacker has to search before the right password is found.

Method	Test Cases		
	password 1	password 2	password 3
Baseline(n -Viterbi)	7.8%	6.6%	6.8%
Timing Averaging	0.38%	0.34%	0.05%
m - n -Viterbi	0.39%	0.34%	0.05%

designed to make it comparable with that of the prior work [26]: we chose 15 keys for training and testing an HMM, which include 13 letters and 2 numbers⁹. From these keys, we identified 225 key pairs and measured 45 inter-keystroke timings for each of these pair from a user. We found that the timing for each pair indeed had Gaussian-like distributions. These distributions were used to parameterize two HMMs: one for the first 4 bytes of an 8-byte password and the other for the second half.

We randomly selected 3 passwords from the space of all possible 8-byte sequences formed by the 15 characters. For each password, we ran the n -Viterbi algorithm on 50 timing sequences. Each of these sequences caused the algorithm to produce a ranking list of candidate passwords. The position of the real password on the list describes the search space an attacker has to explore: for example, we only need to check 1012 candidates if the password is the 1012th member on the list, which reduces the search space for a 4-byte password by 50 times. To avoid the intensive computation, our implementation only output the top 4500 members from an HMM. We found that for about 75% of the sequences tested in our experiment, their corresponding passwords were among these members. In Table 4, we present the averaged percentage of the search space for finding a password.

We tested the timing averaging approach and m - n -Viterbi algorithm described in Section 4.2 with 50 timing sequences for each password, and present the results in Table 4. As the table shows, both approaches achieved significant improvements over the n -Viterbi algorithm: they shrank the search space by factors ranging from 250 to 2000. In contrast, the speed-up factor introduced by the n -Viterbi algorithm was much smaller¹⁰.

We also found that the speed-up factors achieved by our approach, like the prior work [26], depended on the letter pairs the victim chose for her password: if the timing distribution of one pair (Figure 6) is not very close to those of other pairs, it can be more reliably determined, which contributes to a more significant reduction of searching spaces. For example, in Figure 6, a password built on the pairs whose means are around 300 milliseconds is much easier to be inferred than the one composed of the pairs around 100 milliseconds, as the latter pairs are more difficult to distinguish from others with very similar distributions. It is important to note that those distributions actually reflect an individual’s typing

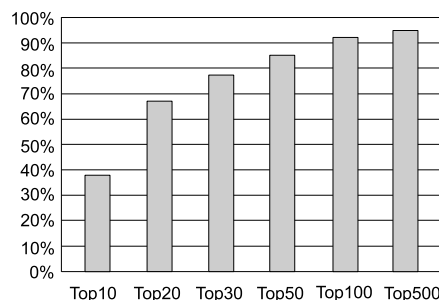


Figure 10: The success rates of the attack on English words

practice, and therefore, the same password entered by one can become easier to crack than by another.

English words. We also studied how the timing information can help infer English words. To prepare for the experiment, a program was used to randomly generate character sequences with lengths of 3, 4 and 5 letters¹¹, and from them, we selected 2103 words that also appeared in a dictionary. These words were classified into three categories according to their lengths. For the words within each category, we computed a distribution using their frequencies reported by [18]. These distributions were used to determine the transition probabilities of the HMMs for individual categories, which we applied to infer the words with different lengths.

In the experiment, we randomly draw words from each category in accordance with their distribution, and typed them to collect timing sequences. The timing segments that represented individual words were identified from the sequences using the feature of the SPACE key. For each segment, we picked up an HMM according to the length of the word and solved it using the n -Viterbi algorithm, which gave us a ranking list of candidates. From the list, our approach further removed the candidates that did not pass a spelling check. We tested 14 3-letter words, 11 4-letter words and 14 5-letter words. The outcomes are described in Figure 10. From the table, we can see that the real words were highly ranked in most cases: almost 40% of them appeared in top 10 and 86% among top 50.

6 Discussion

6.1 Further Study of the Attack

Our current implementation only tracks the call-back function for the key press event. We believe that the pattern for keystroke recognition can be more specific and easier to detect by adding the ESP sequences of the system calls related to the key release event. Moreover, we evaluated our approach using three applications. It is interesting to know whether other common applications

are also subject to our attack. What we learnt from our study is that our attack no longer works when system calls are not immediately triggered by keystrokes. This could happen when the victim's process postpones the necessary actions such as access to the standard I/O until multiple keystrokes are received. For example, `su` does not read a password character by character, and instead, imports the string as a whole; as a result, it cannot be attacked when it is not used under the interactive mode of SSH. As another example, GTK+ applications tend to display keys only when the buffer holding them becomes full or a timer is triggered. Further study to identify the type of applications vulnerable to our attack is left as our future research. In addition, it is conceivable that the same techniques can be applied beyond identification of inter-keystroke timing. For example, we can track the ESP dynamics caused by other events such as moving mouse to peek into a user's activities.

Our current research focuses more on extracting inter-keystroke timings from an application than on analyzing these timings. Certainly more can be done to improve our timing analysis techniques. Specifically, password cracking can be greatly facilitated with the knowledge about the types of individual password characters such as letter or number. Acquisition of such knowledge can be achieved using our enhanced versions of the *n*-Viterbi algorithm that accept multiple timing sequences. This "classification" attack can be more effective than the timing attack proposed in [26], as it does not need to deal with a large key-pair space. Moreover, the approach we used to infer English words is still preliminary. We did not evaluate it using long words, because solving the HMMs for these words can be time consuming. A straightforward solution is to split a long word into small segments and model each of them with an HMM, as we did for password cracking. This treatment, however, could miss the inherent relations between the segments of a word, which is important because letters in a word are often correlated. Fundamentally, the first-order HMM we adopted is limited in its capability of modeling such relations: it cannot describe the dependency relation beyond that between two key pairs. Application of other language models such as the high-order HMM [12] can certainly improve our techniques.

Actually, ESP/EIP is by no means the only information within `procfs` that can be used for acquiring inter-keystroke timings. Other information that can lead to a similar attack includes interrupt statistics file `/proc/interrupts`, and network status data `/proc/net`. The latter enables an attacker to track the activities of the TCP connections related to the inputs from a remote client. Moreover, the `procfs` of most UNIX-like systems expose the *system time* of a process, i.e., the amount of time the kernel spends serving the sys-

tem calls from the process. Disclosure of such information actually enables keystroke eavesdropping, which is elaborated in Section 6.2.

6.2 Information Leaks in the `Procfs` of Other UNIX-like Systems

Besides Linux, most other UNIX-like systems also implement `procfs`. These implementations vary from case to case, and as a result, their susceptibilities to side-channel attacks also differ. Here we discuss such privacy risks on two systems, FreeBSD and OpenSolaris.

FreeBSD manages its process files more cautiously than Linux¹²: it puts all register values into the file `/proc/pid/regs` that can only be read by the owner of a process, which blocks the information used by our attack. However, we found that other information released by the `procfs` can lead to similar attacks. A prominent example is the system time reported by `/proc/pid/status`, a file open to every user. Figure 11 shows the correlations between the time consumed by `vim` and the keystrokes it received, as observed in our research. This demonstrates that keystroke events within the process can be identified from the change of its system time, which makes keystroke eavesdropping possible. A problem here is that we may not be able to detect special keys a user enters, for example, "MOV_CURSOR", which is determined from ESP/EIP information on Linux. A possible solution is using the discrepancies of system-time increments triggered by different keys being entered to fingerprint these individual keys. Further study of this technique is left to our future research.

OpenSolaris kernel makes the `/proc` directory of a process only readable to its owner, which prevents other users from entering that directory. Interestingly, some files under the directory are actually permitted to be read by others, for supporting the applications such as `ps` and `top`. Like FreeBSD, the registers of the process are kept off-limits. However, other information, including system time, is still open for grabs. Figure 11 illustrates the changes of the system time versus a series of keystrokes we entered on OpenSolaris, which demonstrates that identification of inter-keystroke timings is completely feasible on the system.

6.3 Defense

An immediate defense against our attack is to prevent one from reading the `stat` file of another user's process once it is forked, which can be done by manually changing the permissions of the file. However, this approach is not reliable because human are error-prone and whenever the step for altering permissions is inadvertently missed,

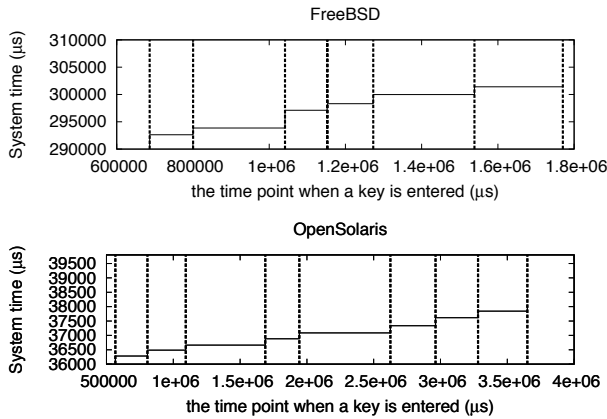


Figure 11: System time (solid line) vs. keystroke events (dashed line) in vim under FreeBSD (Release 7.1) and OpenSolaris (Release 2008.11). In the experiments, we found that the system time of vim changed only in response to keystrokes, which were recorded by shadow programs.

the door to our attack becomes wide open. The approach also affects the normal operations of common tools such as `ps` and `top`, which all depend on `stat` to acquire process information. A complete solution is to patch Linux kernels to remove the ESP and EIP information from a process’s virtual file or move them into a separate file which can only be read by the owner. The problem is that there is no guarantee that other information disclosed by procsfs will not lead to a similar attack (Section 6.1 and Section 6.2). Detection of our attack can also be hard, because our shadow process behaves exactly like the legitimate tools such as `top`, which also continuously read from virtual files. The shadow program can also hide its CPU usage by leveraging existing techniques [32]. Fundamentally, with the pervasiveness of multi-core systems that enable one process to effectively monitor another process’s execution, we feel it is necessary to rethink the security implications of the public information available on current multi-user systems.

7 Related Work

It has long been known that individual users can be characterized by their unique and stable keystroke dynamics, the timing information that can be observed when one is typing [16]. Such information has been intensively studied for biometric authentication [21]. In comparison, little has been done to explore its potential for inferring the characters a user typed [6]. The first paper on this subject¹³ proposes to measure inter-keystroke timings from the latencies between SSH packets [7] and use them to crack passwords. Our attack takes a different path to ac-

quire timings: we take advantage of the information of a process exposed by procsfs to find out when a key is received by the process, which has been made possible by the rapid development of multi-core techniques. Compared with the prior approach, our attack can happen to the clients who use a multi-user system locally as well as those who connect to the system remotely. Moreover, our timing analysis is much more accurate than the prior approach, through effective use of the information available from procsfs. On the downside, we need a user account to launch our attack, which is not required by the prior approach. Another prior proposal measures CPU timings to acquire the information about the password a user enters [31]. This approach only gets the information such as password length and some special characters, and is subject to the interference of the activities such as processing mouse events, whereas our approach can accurately identify the events related to keystrokes and infer the characters being entered. Timing analysis has also been applied to attack cryptosystems [5, 34, 17, 8].

Keyboard acoustic emanations [34] also leak out information regarding a user’s keystrokes. Such information has been leveraged by several prior approaches [2, 33, 3] to identify the keys being entered. Similar to our attack, some of these approaches also apply language models (including the high-order HMM) to infer English words. They all report very high success rates. Acoustic emanations are associated to individual keys, whereas timings are measured between a pair of keys. This makes character inference based on timings more challenging. On the other hand, acquisition of acoustic emanations requires physically implanting a recording device close to the victim, whereas our attack only needs a normal user account. Moreover, these attacks can only be used against a local user. In contrast, our approach works on both local and remote users.

8 Conclusion

In this paper, we present a new attack that allows a malicious user to eavesdrop on other users’ keystrokes using procsfs, a virtual file system that shares statistic information regarding individual users’ processes. Our attack utilizes the stack information of a process present in its `stat` file on a Linux system to fingerprint its behavior when a keystroke is received. Such behavior is modeled as an ESP pattern of its system calls, which can be extracted from an application through automatic program analysis. During the runtime of the application, our approach shadows its process with another process to collect an ESP trace from its `stat` file. Our research shows that on a multi-core system, the shadow process can acquire a trace with a sufficient granularity for identifying keystroke events. This allows us to determine the tim-

ings between keystrokes and analyze them to infer the key sequence the victim entered. We also show that other information available from procfs can be of great help to character inference: knowing that the same user enters her password to the same application, we can combine multiple timing sequences related to the password to significantly reduce the space for searching it. We also propose to utilize the victim's writing style to infer the English words she enters. Both approaches are very effective, according to our experimental study.

Our attack can be further improved through adopting more advanced analysis techniques such as the high-order HMM and other language model. The same idea can also be applied to infer other user activities such as moving and clicking mouse, and even deduce others' secret keys. More generally, other information within procfs, such as system time, can be used for a similar attack, which threatens other UNIX-like systems such as FreeBSD and OpenSolaris. Research in these directions is left as our future work.

Acknowledgements

The authors thank our shepherd Angelos Stavrou for his guidance on the preparation of the final version, and anonymous reviewers for their comments on the draft of the paper. We also thank Rui Wang for his assistance in preparing one of the experiments reported in the paper. This work was supported in part by the National Science Foundation the Cyber Trust program under Grant No. CNS-0716292.

References

- [1] Cryptography/frequency analysis. http://en.wikibooks.org/wiki/Cryptography:Frequency_analysis, Aug 2006.
- [2] ASONOV, D., AND AGRAWAL, R. Keyboard acoustic emanations. In *IEEE Symposium on Security and Privacy* (2004), pp. 3–11.
- [3] BERGER, Y., WOOL, A., AND YEREDOR, A. Dictionary attacks using keyboard acoustics emanations. In *CCS* (2006), ACM, pp. 245–254.
- [4] BERGROTH, L., HAKONEN, H., AND RAITA, T. A survey of longest common subsequence algorithms. In *Proceedings of Seventh International Symposium on String Processing and Information Retrieval* (2000), pp. 39–48.
- [5] BRUMLEY, D., AND BONEH, D. Remote timing attacks are practical. In *In proceedings of the 12th Usenix Security Symposium* (2003).
- [6] BUCHHOLTZ, M., GILMORE, S. T., HILLSTON, J., AND NIELSON, F. Securing statically-verified communications protocols against timing attacks. *Electronic Notes in Theoretical Computer Science* 128, 4 (2005), 123–143.
- [7] DESIGNER, S., AND SONG, D. Passive analysis of ssh (secure shell) traffic. Openwall advisory OW-003, March 2001.
- [8] DHEM, J. F., KOEUNE, F., LEROUX, P.-A., MESTRE, P., QUISQUATER, J.-J., AND WILLEMS, J.-L. A practical implementation of the timing attack. In *Proceedings of CARDIS* (1998), pp. 167–182.
- [9] DISTROWATCH.COM. Top ten distributions: An overview of today's top distributions. <http://distrowatch.com/dwres.php?resource=major>, 2008.
- [10] EDIT VIRTUAL LANGUAGE CENTER. Word frequency lists. <http://www.edict.com.hk/textanalyser/wordlists.htm>, as of September, 2008.
- [11] FERRELL, J. procfs: Gone but not forgotten. <http://www.freebsd.org/doc/en/articles/linux-users/procfs.html>, 2009.
- [12] FRANCOIS, M. J., AND PAUL, H. J. Automatic word recognition based on second-order hidden markov models. In *ICSLP* (1994), pp. 247–250.
- [13] HOGYE, M. A., HUGHES, C. T., SARFATY, J. M., AND WOLF, J. D. Analysis of the feasibility of keystroke timing attacks over ssh connections. Technical Report CS588, School of Engineering and Applied Science, University of Virginia, December 2001.
- [14] INC., R. Process directories. <http://www.redhat.com/docs/manuals/enterprise/RHEL-4-Manual/en-US/ReferenceGuide/s2-proc-processdirs.html>, 2007.
- [15] JONES, N. C., AND PEVZNER, P. A. *An Introduction to Bioinformatics Algorithms*. the MIT Press, August 2004.
- [16] JOYCE, R., AND GUPTA, G. Identity authorization based on keystroke latencies. *Communications of the ACM* 33, 2 (1990), 168–176.
- [17] KOCHER, P., JAE, J., AND JUN, B. Differential power analysis. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology* (1999), Springer-Verlag, pp. 388–397.
- [18] LEECH, G., RAYSON, P., AND WILSON, A. Word frequencies in written and spoken english: based on the british national corpus. <http://www.comp.lancs.ac.uk/ucrel/bncfreq>.
- [19] LOSCOCO, P., AND SMALLEY, S. procfs analysis. <http://www.nsa.gov/SeLinux/papers/slinux/node57.html>, February 2001.
- [20] LUK, C. K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LONEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation* (2005), pp. 190–200.
- [21] MONROSE, F., AND RUBIN, A. Authentication via keystroke dynamics. In *Proceedings of the 4th ACM conference on Computer and communications security* (1997), ACM Press, pp. 48–56.
- [22] PETERSSON, J. What is linux-gate.so.1? <http://www.trilithium.com/johan/2005/08/linux-gate/>, as of September, 2008.
- [23] PROVOS, N. Systrace - interactive policy generation for system calls. <http://www.citi.umich.edu/u/provos/systrace/>, 2006.
- [24] RABINER, L. R. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE* 77, 2 (1989), 257–286.
- [25] SECURITY, S. C. Timing analysis is not a real-life threat to ssh secure shell users. <http://www.ssh.com/company/news/2001/english/all/article/204/>, November 2001.

- [26] SONG, D. X., WAGNER, D., AND TIAN, X. Timing analysis of keystrokes and timing attacks on ssh. In *USENIX Security Symposium* (2001), USENIX Association.
- [27] SOURCEFORGE.NET. <http://sourceforge.net/projects/strace/>, August 2008.
- [28] TEAM, G. <http://www.gtk.org>, as of September, 2008.
- [29] TEAM, P. Pax address space layout randomization (aslr). <http://pax.grsecurity.net/docs/aslr.txt>, March 2003.
- [30] TEAM, P. <http://pax.grsecurity.net/>, as of September, 2008.
- [31] TROSTLE, J. Timing attacks against trusted path. In *IEEE Symposium on Security and Privacy* (1998).
- [32] TSAFRIR, D., ETSION, Y., AND FEITELSON, D. G. Secretly monopolizing the cpu without superuser privileges. In *Proceedings of 16th USENIX Security Symposium* (Berkeley, CA, USA, 2007), USENIX Association, pp. 1–18.
- [33] ZHANG, L., ZHOU, F., AND TYGAR, J. D. Keyboard acoustic emanations revisited. In *CCS'05: ACM Conference on Computer and Communications Security* (2005), ACM Press, pp. 373–382.
- [34] ZHOU, Y., AND FENG, D. Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing. csrc.nist.gov/groups/STM/cmvp/documents/fips140-3/physec/papers/physecpaper19.pdf, December 2005.

Notes

¹The program is actually a simplified version of `vim`.

²Some old Linux distributions such as RedHat Enterprise 4 do not use `vDSO`, and instead then entry of their system calls points to `_dl_sysinfo_int80` in library `/lib/ld-linux.so` or `/lib/ld.so`.

³We designed our attack in a way that a keystroke event can be reliably identified even in the presence of some missing ESP/EIP values, which could happen when the shadow process is preempted by other processes (Section 3).

⁴After the application enter the state that keystroke inputs are expected, our approach waits for a time period before exporting the first sequence. This allows for the accomplishment of all the system calls prior to keystrokes. Similarly, the second sequence is not exported until the keystroke happens for a while so as to ensure that all the system calls related to the stroke are completed.

⁵There are actually two events associated with a keystroke: key press and key release. We use the first event here for the simplicity of explanation. Our technique can actually be applied to both events.

⁶We did not use the instructions such as `'ret'` to identify the end of a call-back function because compiler optimization could remove such instructions from a binary executable.

⁷Some Linux versions such as RedHat [14] turn off the permissions on `maps` but `stat` is always open.

⁸Theoretically, this approach may not eliminate false positives when it comes to non-deterministic applications, because these applications may contain ESP sequences we did not observe during the offline analysis.

⁹The prior work used 10 letters and 5 numbers. We increased the number of letter keys to get a larger set of legitimate words for our experiment on English text.

¹⁰The factor is actually below what was reported in the prior work [26]. A possibility is that we adopted 225 key pairs rather than 142 used in the prior work.

¹¹We did not choose longer words in our experiment to avoid intensive computation. However, such a word can also be learnt through splitting it into shorter segments and analyzing them using different HMMs.

¹²It is reported that FreeBSD moves to phase out `procf`s [11].

¹³The possibility of timing attack on SSH has also been briefly discussed in [26].

A Practical Congestion Attack on Tor Using Long Paths

Nathan S. Evans
*Colorado Research Institute
for Security and Privacy
University of Denver
Email: nevans6@du.edu*

Roger Dingledine
*The Tor Project
Email: arma@mit.edu*

Christian Grothoff
*Colorado Research Institute
for Security and Privacy
University of Denver
Email: christian@grothoff.org*

Abstract

In 2005, Murdoch and Danezis demonstrated the first practical congestion attack against a deployed anonymity network. They could identify which relays were on a target Tor user's path by building paths one at a time through every Tor relay and introducing congestion. However, the original attack was performed on only 13 Tor relays on the nascent and lightly loaded Tor network.

We show that the attack from their paper is no longer practical on today's 1500-relay heavily loaded Tor network. The attack doesn't scale because a) the attacker needs a tremendous amount of bandwidth to measure enough relays during the attack window, and b) there are too many false positives now that many other users are adding congestion at the same time as the attacks.

We then strengthen the original congestion attack by combining it with a novel bandwidth amplification attack based on a flaw in the Tor design that lets us build long circuits that loop back on themselves. We show that this new combination attack is practical and effective by demonstrating a working attack on today's deployed Tor network. By coming up with a model to better understand Tor's routing behavior under congestion, we further provide a statistical analysis characterizing how effective our attack is in each case.

1 Introduction

This paper presents an attack which exploits a weakness in Tor's circuit construction protocol to implement an improved variant of Murdoch and Danezis's congestion attack [26, 27]. Tor [12] is an anonymizing peer-to-peer network that provides users with the ability to establish low-latency TCP tunnels, called circuits, through a network of relays provided by the peers in the network. In 2005, Murdoch and Danezis were able to determine the path that messages take through the Tor network by causing congestion in the network and then observing the changes in the traffic patterns.

While Murdoch and Danezis's work popularized the idea proposed in [1] of an adversary perturbing traffic patterns of a low-latency network to deanonymize its users, the original attack no longer works on the modern Tor network. In a network with thousands of relays, too many relays share similar latency characteristics and the amount of congestion that was detectable in 2005 is no longer significant; thus, the traffic of a single normal user does not leave an easily distinguishable signature in the significantly larger volume of data routed by today's Tor network.

We address the original attack's weaknesses by combining JavaScript injection with a selective and asymmetric denial-of-service (DoS) attack to obtain specific information about the path selected by the victim. As a result, we are able to identify the entire path for a user of today's Tor network. Because our attack magnifies the congestion effects of the original attack, it requires little bandwidth on the part of the attacker. We also provide an improved method for evaluating the statistical significance of the obtained data, based on Tor's message scheduling algorithm. As a result, we are not only able to determine which relays make up the circuit with high probability, we can also quantify the extent to which the attack succeeds. This paper presents the attack and experimental results obtained from the actual Tor network.

We propose some non-trivial modifications to the current Tor protocol and implementation which would raise the cost of the attack. However, we emphasize that a full defense against our attack is still not known.

Just as Murdoch and Danezis's work applied to other systems such as MorphMix [24] or Tarzan [36], our improved attack and suggested partial defense can also be generalized to other networks using onion routing. Also, in contrast to previously proposed solutions to congestion attacks [18, 22–24, 28, 30, 35, 36], our proposed modifications do not impact the performance of the anonymizing network.

2 Related Work

Chaum's mixes [3] are a common method for achieving anonymity. Multiple encrypted messages are sent to a mix from different sources and each is forwarded by the mix to its respective destination. Combinations of artificial delays, changes in message order, message batching, uniform message formats (after encryption), and chaining of multiple mixes are used to further mask the correspondence between input and output flows in various variations of the design [5, 7, 8, 17, 21, 25, 32, 33]. Onion routing [16] is essentially the process of using an initiator-selected chain of low-latency mixes for the transmission of encrypted streams of messages in such a way that each mix only knows the previous and the next mix in the chain, thus providing initiator-anonymity even if some of the mixes are controlled by the adversary.

2.1 Tor

Tor [12] is a distributed anonymizing network that uses onion routing to provide anonymity for its users. Most Tor users access the Tor network via a local proxy program such as Privoxy [20] to tunnel the HTTP requests of their browser through the Tor network. The goal is to make it difficult for web servers to ascertain the IP address of the browsing user. Tor provides anonymity by utilizing a large number of distributed volunteer-run relays (or routers). The Tor client software retrieves a list of participating relays, randomly chooses some number of them, and creates a circuit (a chain of relays) through the network. The circuit setup involves establishing a session key with each router in the circuit, so that data sent can be encrypted in multiple layers that are peeled off as the data travels through the network. The client encrypts the data once for each relay, and then sends it to the first relay in the circuit; each relay successively peels off one encryption layer and forwards the traffic to the next link in the chain until it reaches the final node, the exit router of the circuit, which sends the traffic out to the destination on the Internet.

Data that passes through the Tor network is packaged into fixed-sized cells, which are queued upon receipt for processing and forwarding. For each circuit that a Tor router is a part of, the router maintains a separate queue and processes these queues in a round-robin fashion. If a queue for a circuit is empty it is skipped. Other than using this fairness scheme, Tor does not intentionally introduce any latency when forwarding cells.

The Tor threat model differs from the usual model for anonymity schemes [12]. The traditional threat model is that of a global passive adversary: one that can observe all traffic on the network between any two links. In contrast, Tor assumes a non-global adversary which can only observe some subset of the connections and

can control only a subset of Tor nodes. Well-known attack strategies such as blending attacks [34] require more powerful attackers than those permitted by Tor's attacker model. Tor's model is still valuable, as the resulting design achieves a level of anonymity that is sufficient for many users while providing reasonable performance. Unlike the aforementioned strategies, the adversary used in this paper operates within the limits set by Tor's attacker model. Specifically, our adversary is simply able to run a Tor exit node and access the Tor network with resources similar to those of a normal Tor user.

2.2 Attacks on Tor and other Mixes

Many different attacks on low-latency mix networks and other anonymization schemes exist, and a fair number of these are specifically aimed at the Tor network. These attacks can be broadly grouped into three categories: path selection attacks, passive attacks, and active attacks. Path selection attacks attempt to invalidate the assumption that selecting relays at random will usually result in a safe circuit. Passive attacks are those where the adversary in large part simply observes the network in order to reduce the anonymity of users. Active attacks are those where the adversary uses its resources to modify the behavior of the network; we'll focus here on a class of active attacks known as congestion or interference attacks.

2.2.1 Path Selection Attacks

Path selection is crucial for the security of Tor users; in order to retain anonymity, the initiator needs to choose a path such that the first and last relay in the circuit won't collude. By selecting relays at random during circuit creation, it could be assumed that the probability of finding at least one non-malicious relay would increase with longer paths. However, this reasoning ignores the possibility that malicious Tor routers might choose only to facilitate connections with other adversary-controlled relays and discard all other connections [2]; thus the initiator either constructs a fully malicious circuit upon randomly selecting a malicious node, or fails that circuit and tries again. This type of attack suggests that longer circuits do not guarantee stronger anonymity.

A variant of this attack called "packet spinning" [30] attempts to force users to select malicious routers by causing legitimate routers to time out. Here the attacker builds circular paths throughout the Tor network and transmits large amounts of data through those paths in order to keep legitimate relays busy. The attacker then runs another set of (malicious) servers which would eventually be selected by users because of the attacker-generated load on all legitimate mixes. The attack is successful if, as a result, the initiator chooses only malicious servers for its circuit, making deanonymization trivial.

2.2.2 Passive Attacks

Several passive attacks on mix systems were proposed by Back et al. [1]. The first of these attacks is a “packet counting” attack, where a global passive adversary simply monitors the initiator’s output to discover the number of packets sent to the first mix, then observes the first mix to watch for the same number of packets going to some other destination. In this way, a global passive adversary could correlate traffic to a specific user. As described by Levine et al. [23], the main method of defeating such attacks is to pad the links between mixes with cover traffic. This defense is costly and may not solve the problem when faced with an active attacker with significant resources; an adversary with enough bandwidth can deal with cover traffic by using up as much of the allotted traffic between two nodes as possible with adversary-generated traffic [4]. As a result, no remaining bandwidth is available for legitimate cover traffic and the adversary can still deduce the amount of legitimate traffic that is being processed by the mix. This attack (as well as others described in this context) requires the adversary to have significant bandwidth. It should be noted that in contrast, the adversary described by our attack requires only the resources of an average mix operator.

Low-latency anonymity systems are also vulnerable to more active timing analysis variations. The attack presented in [23] is based on an adversary’s ability to track specific data through the network by making minor timing modifications to it. The attack assumes that the adversary controls the first and last nodes in the path through the network, with the goal of discovering which destination the initiator is communicating with. The authors discuss both correlating traffic “as is” as well as altering the traffic pattern at the first node in order to make correlation easier at the last node. For this second correlation attack, they describe a packet dropping technique which creates holes in the traffic; these holes then percolate through the network to the last router in the path. The analysis showed that without cover traffic (as employed in Tarzan [14, 15]) or defensive dropping [23], it is relatively easy to correlate communications through mix networks. Even with “normal” cover traffic where all packets between nodes look the same, Shmatikov and Wang show that the traffic analysis attacks are still viable [35]. Their proposed solution is to add cover traffic that mimics traffic flows from the initiator’s application.

A major limitation of all of the attacks described so far is that while they work well for small networks, they do not scale and may fail to produce reliable results for larger anonymizing networks. For example, Back’s active latency measuring attack [1] describes measuring the latencies of circuits and then trying to determine the nodes that were being utilized from the latency of a specific circuit. As the number of nodes grows, this attack

becomes more difficult (due to an increased number of possible circuits), especially as more and more circuits have similar latencies.

2.2.3 Congestion Attacks

A more powerful relative of the described timing attacks is the clogging or congestion attack. In a clogging attack, the adversary not only monitors the connection between two nodes but also creates paths through other nodes and tries to use all of their available capacity [1]; if one of the nodes in the target path is clogged by the attacker, the observed speed of the victim’s connection should change.

In 2005, Murdoch and Danezis described an attack on Tor [27] in which they could reveal all of the routers involved in a Tor circuit. They achieved this result using a combination of a circuit clogging attack and timing analysis. By measuring the load of each node in the network and then subsequently congesting nodes, they were able to discover which nodes were participating in a particular circuit. This result is significant, as it reduces Tor’s security during a successful attack to that of a collection of one hop proxies. This particular attack worked well on the fledgling Tor network with approximately fifty nodes; the authors experienced a high success rate and no false positives. However, their clogging attack no longer produces a signal that stands out on the current Tor network with thousands of nodes. Because today’s Tor network is more heavily used, circuits are created and destroyed more frequently, so the addition of a single clogging circuit has less impact. Also, the increased traffic transmitted through the routers leads to false positives or false negatives due to normal network fluctuations. We provide details about our attempt to reproduce Murdoch and Danezis’s work in Section 6.

McLachlan and Hopper [24] propose a similar circuit clogging attack against MorphMix [33], disproving claims made in [36] that MorphMix is invulnerable to such an attack. Because all MorphMix users are *required* to also be mix servers, McLachlan and Hopper achieve a stronger result than Murdoch and Danezis: they can identify not only the circuit, but the user as well.

Hopper et al. [19] build on the original clogging attack idea to construct a network latency attack to guess the location of Tor users. Their attack is two-phase: first use a congestion attack to identify the relays in the circuit, and then build a parallel circuit through those relays to estimate the latency between the victim and the first relay. A key contribution from their work is a more mathematical approach that quantifies the amount of information leaked in bits over time. We also note that without a working congestion attack, the practicality of their overall approach is limited.

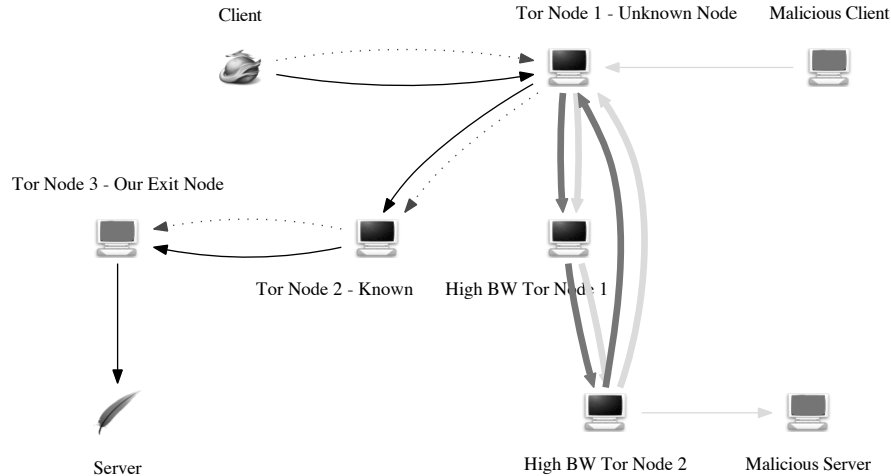


Figure 1: Attack setup. This figure illustrates the normal circuit constructed by the victim to the malicious Tor exit node and the “long” circuit constructed by the attacker to congest the entry (or guard) node used by the victim. The normal thin line from the *client* node to the *server* represents the victim circuit through the Tor network. The unwitting client has chosen the exit server controlled by the adversary, which allows the JavaScript injection. The double thick lines represent the long circular route created by the *malicious client* through the first Tor router chosen by the client. The dotted line shows the path that the JavaScript pings travel.

3 Our Attack

Three features of Tor’s design are crucial for our attack. First of all, Tor routers do not introduce any artificial delays when routing requests. As a result, it is easy for an adversary to observe changes in request latency. Second, the addresses of all Tor routers are publicly known and easily obtained from the directory servers. Tor developers are working on extensions to Tor (called bridge nodes [10, 11]) that would invalidate this assumption, but this service was not widely used at the time of this writing. Finally, the latest Tor server implementation that was available at the time we concluded our original attacks (Tor version 0.2.0.29-rc) did not restrict users from establishing paths of arbitrary length, meaning that there was no restriction in place to limit constructing long paths through Tor servers.¹ We used a modified client version (based on 0.2.0.22-rc) which used a small fixed path length (specifically three) but modified it to use a variable path length specified by our attacker.

Fig. 1 illustrates the three main steps of our attack. First, the adversary needs to ensure that the initiator repeatedly performs requests at known intervals. Second, the adversary observes the pattern in arrival times of these requests. Finally, the adversary changes the pattern by selectively performing a novel clogging attack on

Tor routers to determine the entry node. We will now describe each of these steps in more detail.

3.1 JavaScript Injection

Our attack assumes that the adversary controls an exit node which is used by the victim to access an HTTP server. The attacker uses the Tor exit node to inject a small piece of JavaScript code (shown in Fig. 2) into an HTML response. It should be noted that most Tor users do **not** disable JavaScript and that the popular Tor Button plugin [31] and Privoxy [20] also do not disable JavaScript code; doing so would prevent Tor users from accessing too many web pages. The JavaScript code causes the browser to perform an HTTP request every second, and in response to each request, the adversary uses the exit node to return an empty response, which is thrown away by the browser. Since the JavaScript code may not be able to issue requests precisely every second, it also transmits the local system time (in milliseconds) as part of the request. This allows the adversary to determine the time difference between requests performed by the browser with sufficient precision. (Clock skew on the systems of the adversary and the victim is usually insignificant for the duration of the attack.) While JavaScript is not the only conceivable way for an attacker to cause a browser to transmit data at regular intervals (alternatives include HTTP headers like `refresh` [13]

¹Tor version 0.2.1.3-alpha and later servers restrict path lengths to a maximum of eight because of this work.

```

<script language="javascript">
var count,timer,xmlhttp = 0;
function runonce() {
  xmlhttp = new XMLHttpRequest(); }
function start() {
  xmlhttp.abort();
  xmlhttp = new XMLHttpRequest();
  count++;
  if (timer) clearTimeout(timer);
  timer = setTimeout("start()", 1000);
  myDate = new Date();
  xmlhttp.open("GET",
    "/reportIn.html?num=" + count +
    "&time=" + myDate.getTime(),true);
  xmlhttp.send("");
}
</script>

```

Figure 2: JavaScript code injected by the adversary’s exit node. Note that other techniques such as HTML refresh, could also be used to cause the browser to perform periodic requests.

and HTML images [19]), JavaScript provides an easy and generally rather dependable method to generate such a signal.

The adversary then captures the arrival times of the periodic requests performed by the browser. Since the requests are small, an idle Tor network would result in the differences in arrival times being roughly the same as the departure time differences — these are known because they were added by the JavaScript as parameters to the requests. Our experiments suggest that this is often true for the real network, as most routers are not seriously congested most of the time. This is most likely in part due to TCP’s congestion control and Tor’s built-in load balancing features. Specifically, the variance in latency between the periodic HTTP requests without an active congestion attack is typically in the range of 0–5 s.

However, the current Tor network is usually not entirely idle and making the assumption that the victim’s circuit is idle is thus not acceptable. Observing congestion on a circuit is not enough to establish that the node under the congestion attack by the adversary is part of the circuit; the circuit may be congested for other reasons. Hence, the adversary needs to also establish a baseline for the congestion of the circuit without an active congestion attack. Establishing measurements for the baseline is done before and after causing congestion in order to ensure that observed changes during the attack are caused by the congestion attack and not due to unrelated changes in network characteristics.

The attacker can repeatedly perform interleaved mea-

surements of both the baseline congestion of the circuit and the congestion of the circuit while attacking a node presumed to be on the circuit. The attacker can continue the measurements until either the victim stops using the circuit or until the mathematical analysis yields a node with a substantially higher deviation from the baseline under congestion compared to all other nodes. Before we can describe details of the mathematical analysis, however, we have to discuss how congestion is expected to impact the latency measurements.

3.2 Impact of Congestion on Arrival Times

In order to understand how the congestion attack is expected to impact latency measurements, we first need to take a closer look at how Tor schedules data for routing. Tor makes routing decisions on the level of fixed-size *cells*, each containing 512 bytes of data. Each Tor node routes cells by going round-robin through the list of all circuits, transmitting one packet from each circuit with pending data (see Fig. 3). Usually the number of (active) circuits is small, resulting in little to no delay. If the number of busy circuits is large, messages may start to experience significant delays as the Tor router iterates over the list (see Fig. 4).

Since the HTTP requests transmitted by the injected JavaScript code are small (~250 bytes, depending on count and time), more than one request can fit into a single Tor cell. As a result multiple of these requests will be transmitted at the same time if there is congestion at a router. A possible improvement to our attack would be to use a lower level API to send the packets, as the XMLHttpRequest object inserts unnecessary headers into the request/response objects.

We will now characterize the network’s behavior under congestion with respect to request arrival times. Assuming that the browser transmits requests at a perfectly steady rate of one request per second, a congested router introducing a delay of (at most) n seconds would cause groups of n HTTP requests to arrive with delays of approximately 0, 1, . . . , $n - 1$ seconds respectively: the first cell is delayed by $n - 1$ seconds, the cell arriving a second later by $n - 2$ seconds, and the n -th cell arrives just before the round-robin scheduler processes the circuit and sends all n requests in one batch. This characterization is of course a slight idealization in that it assumes that n is small enough to allow all of the HTTP requests to be grouped into one Tor cell and that there are no other significant fluctuations. Furthermore, it assumes that the amount of congestion caused by the attacker is perfectly steady for the duration of the time measurements, which may not be the case. However, even without these idealizations it is easy to see that the resulting latency histograms would still become “flat” (just not as perfectly

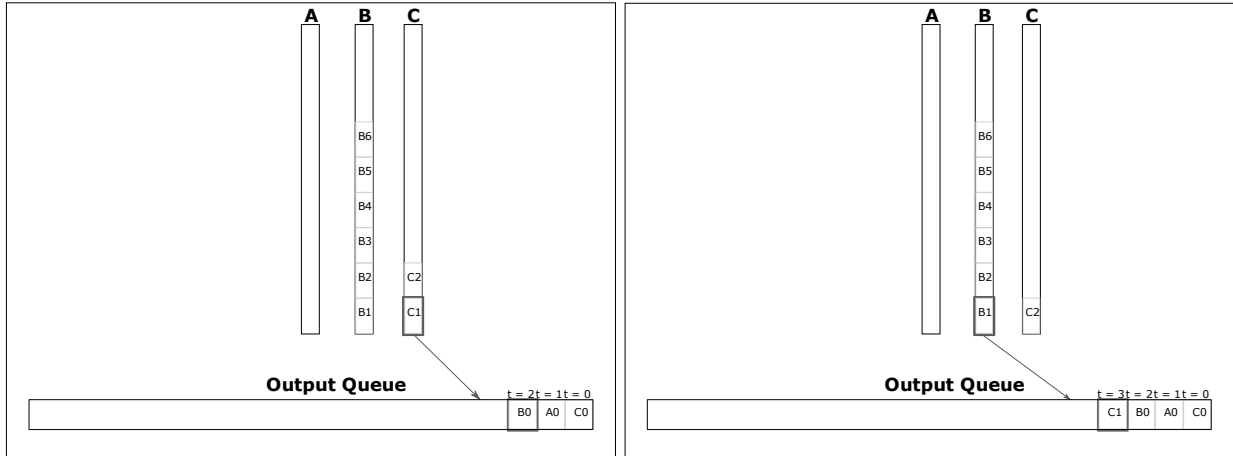


Figure 3: This example illustrates a Tor router which currently is handling three circuits at two points in time ($t = 3$ and $t = 4$). Circuits (A, B and C) have queues; cells are processed one at a time in a round-robin fashion. As the number of circuits increases, the time to iterate over the queues increases. The left figure shows the circuit queues and output queue before selection of cell C1 for output and the right figure shows the queues after queueing C1 for output. The thicker bottom box of queue C (left) and queue B (right) shows the current position of the round-robin queue iterator. At time $t = 1$ the last cell from queue A was processed leaving the queue A empty. As a result, queue A is skipped after processing queue C.

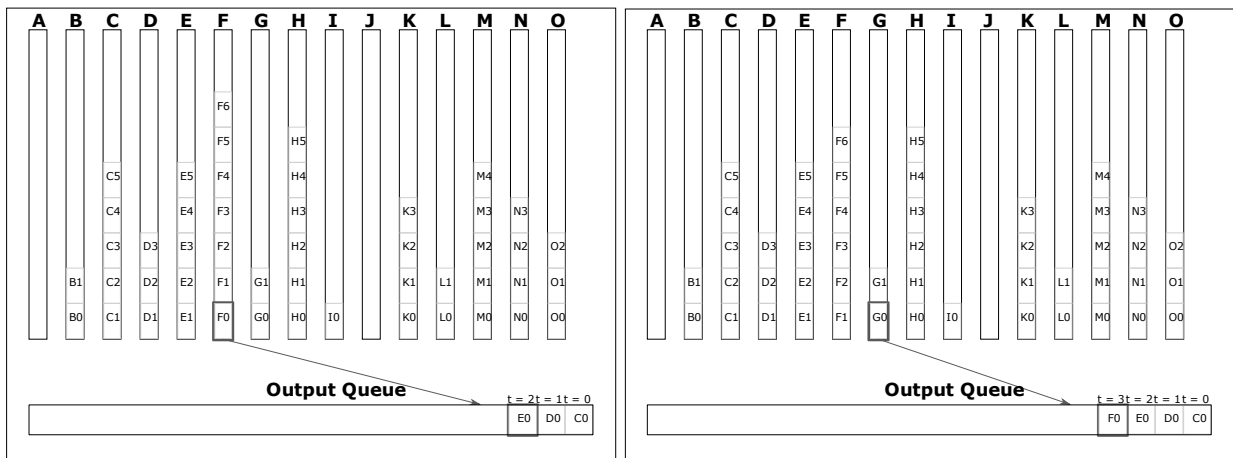


Figure 4: This example illustrates a Tor router under congestion attack handling 15 circuit queues. Note that if a circuit includes a node multiple times, the node assigns the circuit multiple circuit queues. In this example, not all of the circuit queues are busy — this may be because the circuits are not in use or because other routers on the circuit are congested. As in Fig. 3, the left and right figures show the state of the mix before and after queueing a cell, in this case F0.

regular in terms of arrival patterns) assuming the load caused by the attacker is sufficiently high.

Since we ideally expect delays in message arrival times for a congested circuit to follow a roughly flat distribution between zero and n , it makes sense to compute a histogram of the delays in message arrival times. If the congestion attack is targeting a node on the circuit, we would expect to see a roughly equal number of messages in each interval of the histogram. We will call the shape of the resulting histogram *horizontal*. If the circuit is not congested, we expect to see most messages arrive without significant delay which would place them in the bucket for the lowest latency. We will call the shape of the resulting histogram *vertical*. So for example, in Fig. 6 the control data are vertical, whereas the attack data are more horizontal.

Note that the clock difference between the victim's system and the adversary as well as the minimal network delay are easily eliminated by normalizing the observed time differences. As a result, the latency histograms should use the increases in latency over the smallest observed latency, not absolute latencies.

3.3 Statistical Evaluation

In order to numerically capture congestion at nodes we first measure the node's *baseline* latency, that is latency without an active congestion attack (at least as far as we know). We then use the observed latencies to create n bins of latency intervals such that each bin contains the same number of data points. Using the χ^2 -test we could then determine if the latency pattern at the respective peer has changed "significantly". However, this simplistic test is insufficient. Due to the high level of normal user activity, nodes frequently do change their behavior in terms of latencies, either by becoming congested or by congestion easing due to clients switching to other circuits. For the attacker, congestion easing (the latency histogram getting more vertical) is exactly the opposite of the desired effect. Hence the ordinary χ^2 test should not be applied without modification. What the attacker is looking for is the histogram becoming more horizontal, which for the distribution of the bins means that there are fewer values in the low-latency bins and more values in the high-latency bins. For the medium-latency bins no significant change is expected (and any change there is most likely noise).

Hence we modify our computation of the χ^2 value such that we only include changes in the anticipated direction: for the bins corresponding to the lowest third of the latencies, the square of the difference between expected and observed number of events is only counted in the summation if the number of observed events is lower than expected. For the bins corresponding to the high-

est third of the latencies, the square of the difference between expected and observed number of events is only counted if the number of observed events is higher than expected. Since changes to the bins in the middle third are most likely noise, those bins are not included in the χ^2 calculation at all (except as a single additional degree of freedom).

Using this method, a single iteration of measuring the baseline and then determining that there was a significant increase in latency (evidenced by a large χ^2 -value), only signifies that congestion at the guard for the victim circuit was correlated (in time) with the congestion caused by the attacker. Naturally, correlation does not imply causality; in fact, for short (30–60 s) attack runs it frequently happens that the observed χ^2 -value is higher for some false-positive node than when attacking the correct guard node. However, such accidental correlations virtually never survive iterated measurements of the latency baseline and χ^2 -values under attack.

3.4 Congestion Attack

Now we focus on how the attacker controlling the exit node of the circuit will cause significant congestion at nodes that are suspected to be part of the circuit. In general, we will assume that all Tor routers are suspects and that in the simplest case, the attacker will iterate over all known Tor routers with the goal of finding which of these routers is the entry point of the circuit.

For each router X , the attacker constructs a long circuit that repeatedly includes X on the path. Since Tor relays will tear down a circuit that tries to extend to the previous node, we have to use two (or more) other (preferably high-bandwidth) Tor routers before looping back to X . Note that the attacker could choose two different (involuntary) helper nodes in each loop involving X . Since X does not know that the circuit has looped back to X , Tor will treat the long attack circuit as many different circuits when it comes to packet scheduling (Fig. 4).

Once the circuit is sufficiently long (we typically found 24 hops to be effective, but in general this depends on the amount of congestion established during the baseline measurements), the attacker uses the circuit to transmit data. Note that a circuit of length m would allow an attacker with p bandwidth to consume $m \cdot p$ bandwidth on the Tor network, with X routing as much as $\frac{m \cdot p}{3}$ bandwidth. Since X now has to iterate over an additional $\frac{m}{3}$ circuits, this allows the attacker to introduce large delays at this specific router. The main limitation for the attacker here is time. The larger the desired delay d and the smaller the available attacker bandwidth p the longer it will take to construct an attack circuit of sufficient length m : the number of times that the victim node is part of the attack circuit is proportional to the length of

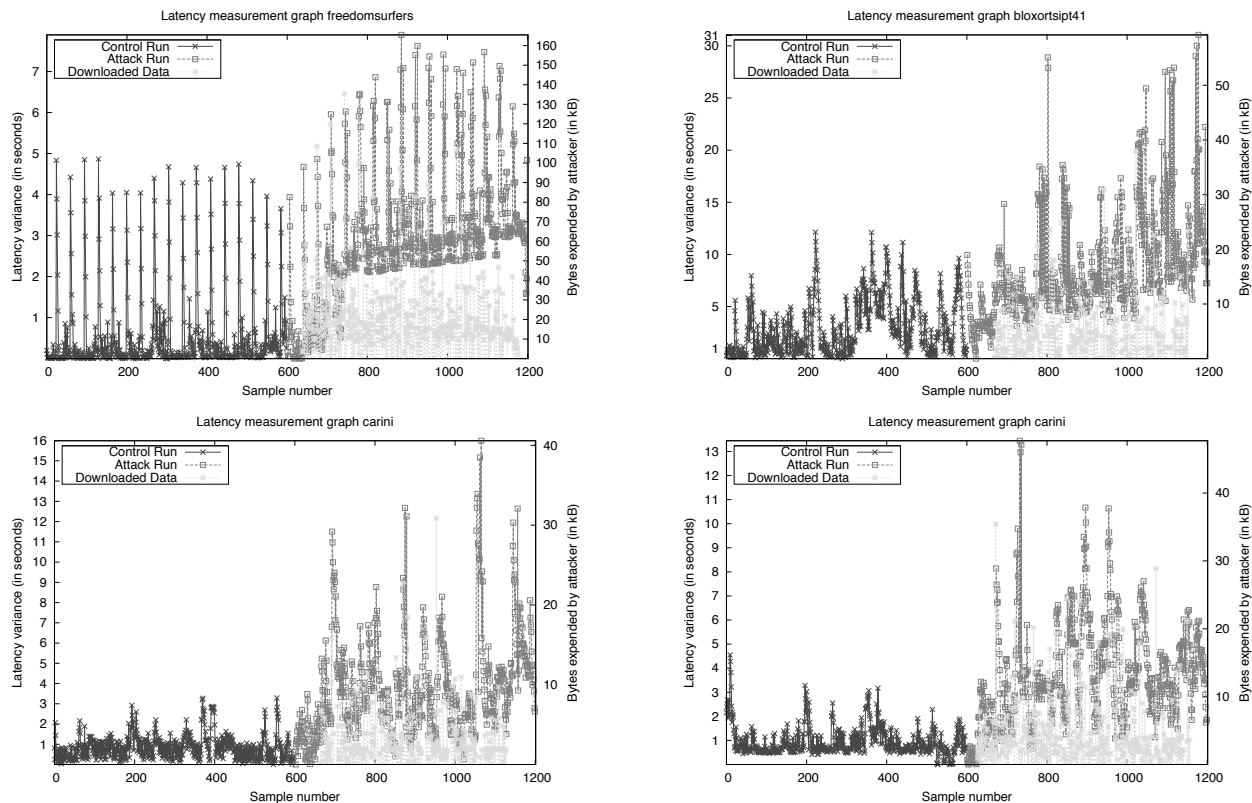


Figure 5: These figures show the results of perturbation of circuits in Tor and the resulting effects on latency. The x -axes show sample numbers (one per second), and the (left) y -axes are latency variance observed on the circuits in seconds. The attack on the first router of each circuit starts at time 600; the third line shows the amount of data (scaled) that transferred through the attack circuit (scaled to the right y -axes). These are individual trials; each shows a single control run and a single attack run.

the circuit m . In other words, the relationship between p , m and the delay d is $d \sim p \cdot m$.

If the router X is independent of the victim circuit, the measured delays should not change significantly when the attack is running. If X is the entry node, the attacker should observe a delay pattern that matches the power of the attack – resulting in a horizontal latency variance histogram as described in Section 3.2. The attacker can vary the strength of the attack (or just switch the long attack circuit between idle and busy a few times) to confirm that the victim’s circuit latency changes correlate with the attack. It should be noted that the attacker should be careful to not make the congestion attack too powerful, especially for low-bandwidth targets. In our experiments we sometimes knocked out routers (for a while) by giving them far too much traffic. As a result, instead of receiving requests from the JavaScript code with increasing latencies, the attacker suddenly no longer receives requests at all, which gives no useful data for the statistical evaluation.

3.5 Optimizations

The adversary can establish many long circuits to be used for attacks before trying to deanonymize a particular victim. Since idle circuits would not have any impact on measuring the baseline (or the impact of using another attack circuit), this technique allows an adversary to eliminate the time needed to establish circuits. As users can only be expected to run their browser for a few minutes, eliminating this delay may be important in practice; even users that may use their browser for hours are likely to change between pages (which might cause Tor to change exit nodes) or disable Tor.

In order to further speed up the process, an adversary can try to perform a binary search for X by initially running attacks on half of the routers in the Tor network. With pre-built attack circuits adding an almost unbounded multiplier to the adversary’s resources, it is conceivable that a sophisticated attacker could probe a network of size s in $\log_2 s$ rounds of attacks.

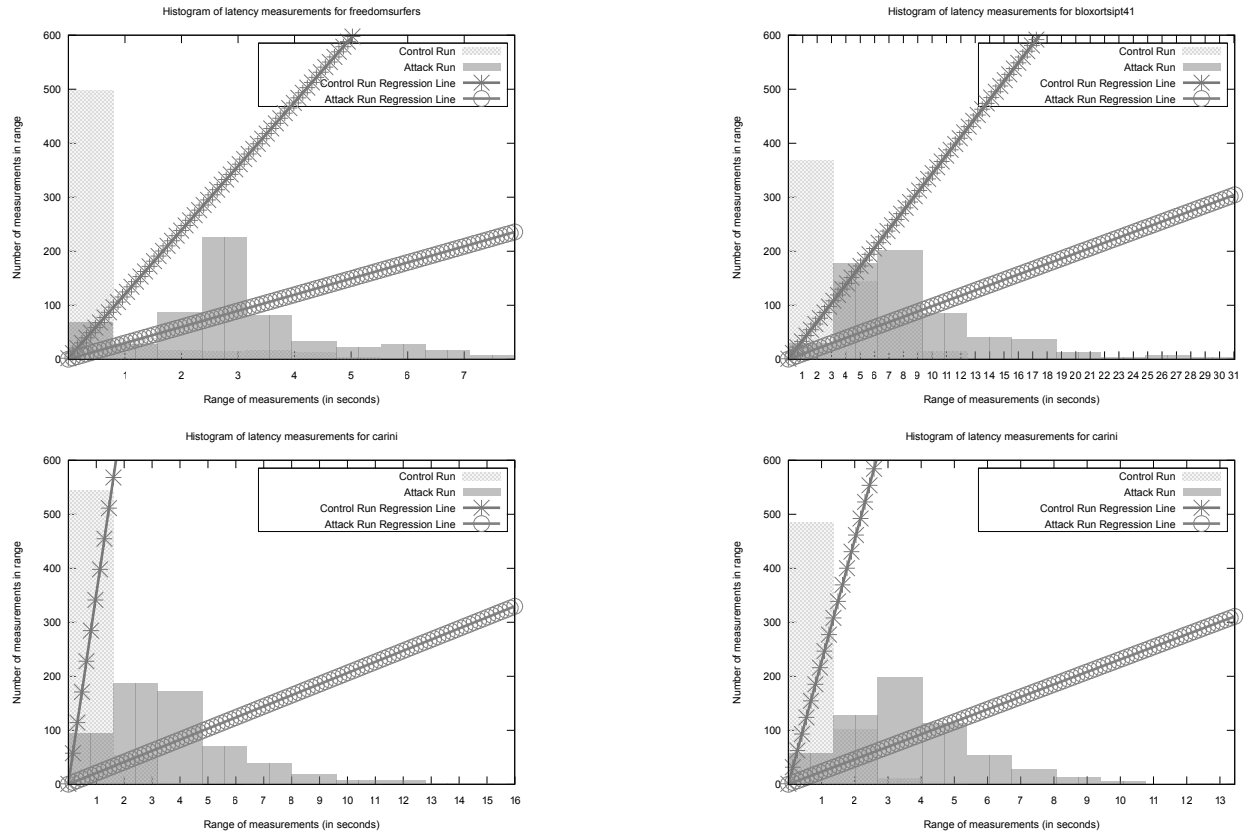


Figure 6: These figures show the results of four independent runs of our congestion attack. In the histograms the x -axis groups ranges of latency variance values together and the y -axis represents the number of readings received in that range. The hash marked bars represent the unperturbed measurements on a circuit and the plain bars show measurements from the same circuit during the attack. The effect of the attack is a shift to higher latency values. The first and second lines are linear least squares fit approximations for the baseline and congestion runs, respectively. These data show the difference between a single control/attack run and are not averages of many runs.

In practice, pre-building a single circuit that would cause congestion for half the network is not feasible; the Tor network is not stable enough to sustain circuits that are thousands of hops long. Furthermore, the differences in available bandwidth between the routers complicates the path selection process. In practice, an adversary would most likely pre-build many circuits of moderate size, forgoing some theoretical bandwidth and attack duration reductions for circuits that are more reliable. Furthermore, the adversary may be able to exclude certain Tor routers from the set of candidates for the first hop based on the overall round-trip latency of the victim's circuit. The Tor network allows the adversary to measure the latency between any two Tor routers [19, 27]; if the overall latency of the victim's circuit is smaller than the latency between the known second router on the path and another router Y , then Y is most likely not a candidate for the entry point.

Finally, the adversary needs to take into consideration that by default, a Tor user switches circuits every 10 minutes. This further limits the window of opportunity for the attacker. However, depending on the browser, the adversary may be able to cause the browser to pipeline HTTP requests which would not allow Tor to switch circuits (since the HTTP session would not end). Tor's circuit switching also has advantages for the adversary: every 10 minutes there is a new chance that the adversary-controlled exit node is chosen by a particular victim. Since users only use a small number of nodes for the first node on a circuit (these nodes are called guard nodes [29]), the adversary has a reasonable chance over time to determine these guard nodes. Compromising one of the guard nodes would then allow full deanonymization of the target user.

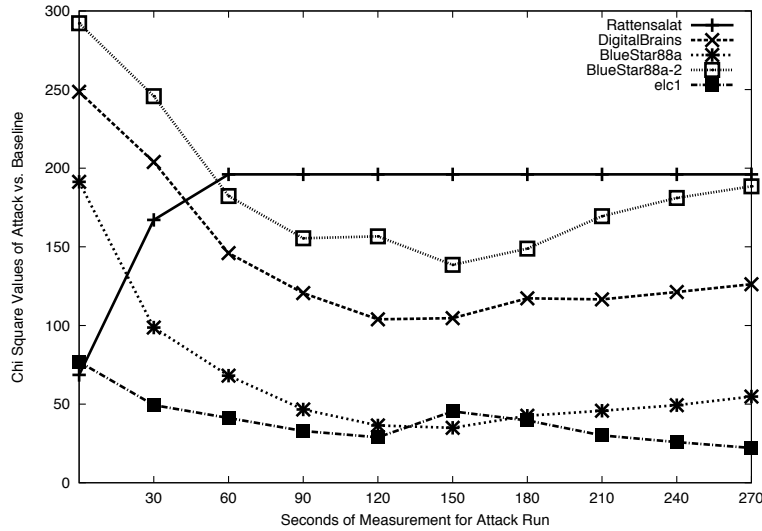


Figure 7: This figure shows the development of χ^2 values (using the modified χ^2 calculation as described in Section 3.3) for the various candidates over a prolonged period of performing a congestion attack on the various nodes. The χ^2 values are computed against a five-minute baseline obtained just prior to the congestion attack. The χ^2 value of the correct entry node quickly rises to the top whereas the χ^2 values for all of the other candidates are typically lower after about a minute of gathering latency information under congestion. This illustrates that a few minutes are typically sufficient to obtain a meaningful χ^2 value.

4 Experimental Results

The results for this paper were obtained by attacking Tor routers on the real, deployed Tor network (initial measurements were done during the Spring and Summer of 2008; additional data was gathered in Spring 2009 with an insignificantly modified attacker setup; the modifications were needed because our original attack client was too outdated to work with the majority of Tor routers at the time). In order to confirm the accuracy of our experiments and avoid ethical problems, we did not attempt to deanonymize real users. Instead, we established our own client circuits through the Tor network to our malicious exit node and then confirmed that our statistical analysis was able to determine the entry node used by our own client. Both the entry nodes and the second nodes on the circuits were normal nodes in the Tor network outside of our control.

The various roles associated with the adversary (exit node, malicious circuit client, and malicious circuit web-server) as well as the “deanonymized” victim were distributed across different machines in order to minimize interference between the attacking systems and the targeted systems. For the measurements we had the simulated victim running a browser requesting and executing the malicious JavaScript code, as well as a machine running the listening server to which the client transmits the “ping” signal approximately every second (Fig. 1). The

browser always connected to the same unmodified Tor client via Privoxy [20]. The Tor client used the standard configuration except that we configured it to use our malicious exit node for its circuits. The other two nodes in the circuit were chosen at random by Tor. Our malicious exit node participated as a normal Tor router in the Tor network for the duration of the study (approximately six weeks). For our tests we did not actually make the exit server inject the JavaScript code; while this is a relatively trivial modification to the Tor code we used a simplified setup with a webserver serving pages with the JavaScript code already present.

The congestion part of the attack requires three components: a simple HTTP server serving an “infinite” stream of random data, a simple HTTP client downloading this stream of data via Tor, and finally a modified Tor client that constructs “long” circuits through those Tor nodes that the attacker would like to congest. Specifically, the modified Tor client allows the attacker to choose two (or more) routers with high bandwidth and a specific target Tor node, and build a long circuit by repeatedly alternating between the target node and the other high bandwidth nodes. The circuit is eventually terminated by connecting from some high-bandwidth exit node to the attacker’s HTTP server which serves the “infinite” stream of random data as fast as the network can process it. As a result, the attacker maximizes the utilization of the Tor circuit. Naturally, an attacker with

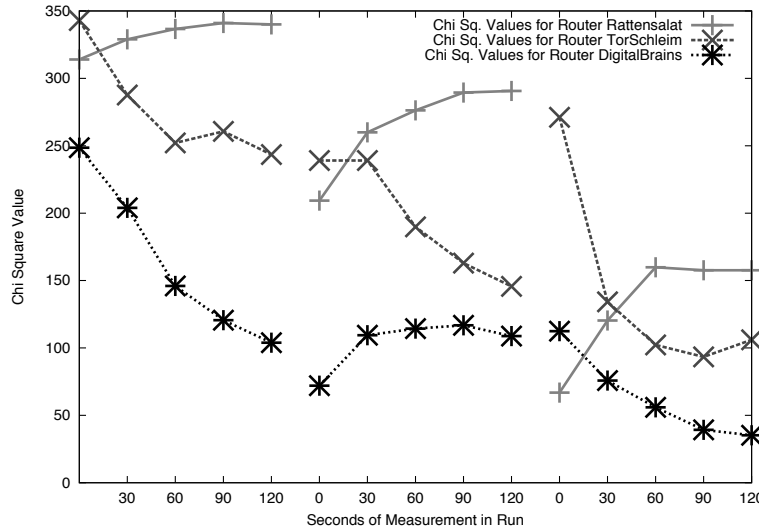


Figure 8: This graph shows three sets of cumulative χ^2 computations for three nodes; the actual entry node (Rattensalat), a node that initially shows up as a false-positive (TorSchleim) and a typical negative (DigitalBrains). As expected, the χ^2 values (at time 120 s) are consistently the highest for the correct node; false-positives can be ruled out through repeated measurements.

significant bandwidth can elect to build multiple circuits in parallel or build shorter circuits and still exhaust the bandwidth resources of the target Tor router.

In order to cause congestion, we simply started the malicious client Tor process with the three chosen Tor routers and route length as parameters and then attempted to connect via `libcurl` [6] to the respective malicious server process. The amount of data received was recorded in order to determine bandwidth consumed during the tests. In order to further increase the load on the Tor network the experiments presented actually used two identical attacker setups with a total of six machines duplicating the three machine setup described in the previous paragraph. We found path lengths of 24 (making our attack strength eight times the attacker bandwidth) sufficient to alter latencies. The overall strength of the attack was measured by the sum of the number of bytes routed through the Tor network by both attacker setups. For each trial, we waited to receive six hundred responses from the “victim”; since the browser transmitted requests to Tor at roughly one request per second, a trial typically took approximately ten minutes.

In addition to measuring the variance in packet arrival time while congesting a particular Tor router, each trial also included baseline measurements of the “un-congested” network to discover the normal variance in packet arrival time for a particular circuit. As discussed earlier, these baseline measurements are crucial for determining the significance of the effect that the congestion attack has had on the target circuit.

Fig. 5 illustrates how running the attack on the first hop of a circuit changes the latency of the received HTTP requests generated by the JavaScript code. The figure uses the same style chosen by Murdoch and Danezis [27], except that an additional line was added to indicate the strength of the attack (as measured by the amount of traffic provided by the adversary). For comparison, the first half of each of the figures shows the node latency variance when it is *not* under active congestion attack (or at least not by us).

While the plots in Fig. 5 visualize the impact of the congestion attack in a simple manner, histograms showing the variance in latency are more suitable to demonstrate the significance of the statistical difference in the traffic patterns. Fig. 6 shows the artificial delay experienced by requests traveling through the Tor network as observed by the adversary. Since Tor is a low-latency anonymization service, the requests group around a low value for a circuit that is not under attack. As expected, if the entry node is under attack, the delay distribution changes from a steep vertical peak to a mostly horizontal distribution. Fig. 6 also includes the best-fit linear approximation functions for the latency histograms which we use to characterize how vertical or how horizontal the histogram is as described in Section 3.2.

Fig. 7 illustrates how the χ^2 values evolve for various nodes over time. Here, we first characterized the baseline congestion for the router for five minutes. Then, the congestion attack was initiated (congesting the various guard nodes). For each attacked node, we used the modified

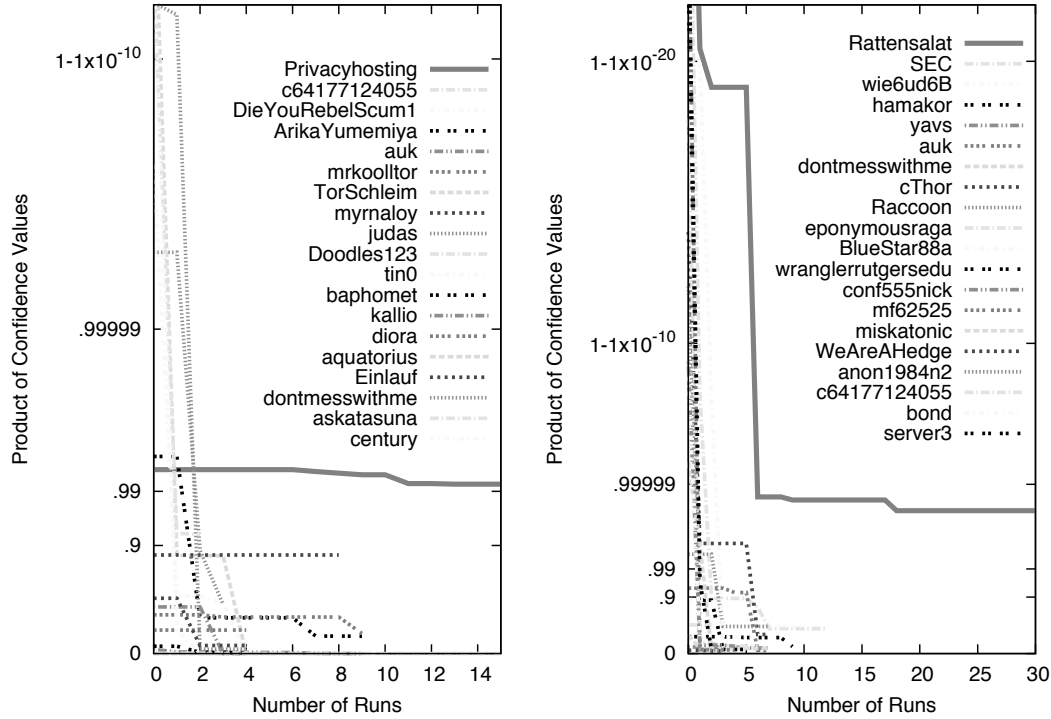


Figure 9: Plot of the product of χ^2 p -values for the top 20 candidate nodes (out of ~ 200 and ~ 250 , respectively) by run (a run is 300 s baseline vs. 300 s attack) for two entry nodes. The first hop (Privacyhosting (left), Rattensalat (right)) routers were tested many more times than other routers, because the others quickly fell to low values. We expect an attacker to perform more measurements for routers that score high to validate the correct entry node was found. Our measurements demonstrate that the multiplied p -value remains consistently high for the correct entry node. The y -axis is plotted on a log scale from 0 to $1 - 1 \times 10^{-10}$ and $1 - 1 \times 10^{-20}$, respectively. We speculate that the lower maximum value for Privacyhosting is due to its higher bandwidth (900 kB/s vs. 231 kB/s).

χ^2 summation (from Section 3.3) to determine how congested the victim’s circuit had become at that time. We computed (cumulative) χ^2 values after 30 s, 60 s, 90 s and so forth. For the χ^2 calculations, we used 60 bins for 300 baseline values; in other words, the time intervals for the bins were chosen so that each bin contained five data points during the five minutes of baseline measurement. The 20 bins in the middle were not included in the summation, resulting in 40 degrees of freedom. As expected, given only 30 s of attack data some “innocent” nodes have higher χ^2 values compared to the entry node (false-positives). However, given more samples the χ^2 values for those nodes typically drop sharply whereas the χ^2 value when congesting the entry node increases or remains high. Of course, false-positive nodes χ^2 values may increase due to network fluctuations over time as well.

Unlucky baseline measurements and shifts in the baseline latency of a router over time can be addressed by iterating between measuring baseline congestion and at-

tack measurements. Fig. 8 shows three iterations of first determining the current baseline and then computing χ^2 values under attack. Again the correct entry node exhibits the largest χ^2 values each time after about a minute of gathering latency data under attack.

Given the possibility of false-positives showing up initially when computing χ^2 values, the attacker should target “all” suspected guard nodes for the first few iterations, and then focus his efforts on those nodes that scored highly. Fig. 9 illustrates this approach. It combines the data from multiple iterations of baseline measurements and χ^2 calculations from attack runs. The attacker determines for each χ^2 value the corresponding confidence interval. These values are frequently large (99.9999% or higher are not uncommon) since Tor routers do frequently experience significant changes in congestion. Given these individual confidence values for each individual iteration, a cumulative score is computed as the product² of these values. Fig. 9 shows the Tor

²It is conceivable that multiplying χ^2 values may cause false-

Router	Πp	r	Peak BW	Configured BW
Rattensalat	0.999991	44	231 kB/s	210 kB/s
c64177124055	0.903	3	569 kB/s	512 kB/s
Raccoon	0.891	8	3337 kB/s	4100 kB/s
wie6ud6B	0.890	11	120 kB/s	100 kB/s
SEC	0.870	13	4707 kB/s	5120 kB/s
cThor	0.789	8	553 kB/s	500 kB/s
BlueStar88a	0.734	7	111 kB/s	100 kB/s
bond	0.697	3	407 kB/s	384 kB/s
eponymousraga	0.458	7	118 kB/s	100 kB/s
conf555nick	0.450	5	275 kB/s	200 kB/s

Table 1: This table lists the top ten (out of 251 total) products of confidence intervals (p -values). r is the number of iterations (and hence the number of factors in Πp) that was performed for the respective router. As expected, the entry node Rattensalat achieves the highest score.

routers with the highest cumulative scores using this metric from trials on two different entry nodes. Note that fewer iterations were performed for routers with low cumulative scores; the router with the highest score (after roughly five iterations) and the most overall iterations is the correctly identified entry node of the circuit.

Table 1 contrasts the product of χ^2 values (as introduced in Section 3.3) obtained while attacking the actual first hop with the product while attacking other Tor routers. The data shows that our attack can be used to distinguish the first hop from other routers when controlling the exit router (therefore knowing a priori the middle router).

Finally, by comparing the highest latency observed during the baseline measurement with the highest latency observed under attack, Table 2 provides a simple illustration showing that the congestion attack actually has a significant effect.

5 Proposed Solutions

An immediate workaround that would address the presented attack would be disabling of JavaScript by the end user. However, JavaScript is not the only means by which an attacker could obtain timing information. For example, redirects embedded in the HTML header could also be used (they would, however, be more visible to the end user). Links to images, frames and other features of HTML could also conceivably be used to generate repeated requests. Disabling all of these features has the disadvantage that the end user’s browsing experience would suffer.

negatives should a single near-zero χ^2 value for the correct entry node be observed. While we have not encountered this problem in practice, using the mean of χ^2 values would provide a way to avoid this theoretical problem.

A better solution would be to thwart the denial-of-service attack inherent in the Tor protocol. Attackers with limited bandwidth would then no longer be able to significantly impact Tor’s performance. Without the ability to selectively increase the latency of a particular Tor router, the resulting timing measurements would most likely give too many false positives. We have extended the Tor protocol to limit the length of a path. The details are described in [9]; we will detail the key points here.

In the modified design, Tor routers now must keep track of how often each circuit has been extended and refuse to route messages that would extend the circuit beyond a given threshold t . This can be done by tagging messages that *may* extend the circuit with a special flag that is not part of the encrypted stream. The easiest way to do this is to introduce a new Tor cell type that is used to flag cells that may extend the circuit. Routers then count the number of messages with the special flag and refuse to route more than a given small number (at the moment, eight) of those messages. Routers that receive a circuit-extension request check that the circuit-extension message is contained in a cell of the appropriate type. Note that these additional checks do not change the performance characteristics of the Tor network. An attacker could still create a long circuit by looping back to an adversary-controlled node every t hops; however, the adversary would then have to provide bandwidth to route every t -th packet; as a result, the bandwidth consumption by the attacker is still bounded by the small constant t instead of the theoretically unbounded path length m .

While this change prevents an attacker from constructing a circuit of arbitrary length, it does not fully prevent the attacker from constructing a path of arbitrary length. The remaining problem is that the attacker could establish a circuit and then from the exit node reconnect to the Tor network again as a client. We could imagine config-

Router Attacked	Max Latency Difference	Avg. Latency Difference	Runs
Rattensalat	70352 ms	25822 ms	41
Wiia	46215 ms	470 ms	5
downtownzion	39522 ms	2625 ms	9
dontmesswithme	37648 ms	166 ms	8
wie6ud6B	35058 ms	9628 ms	9
TorSchleim	28630 ms	5765 ms	15
hamakor	25975 ms	6532 ms	8
Vault24	24330 ms	4647 ms	7
Einlauf	22626 ms	2017 ms	8
grsrlfz	22545 ms	10112 ms	2

Table 2: This table shows the top 10 highest latency differences between the maximum observed measurement in attack runs versus the baseline runs for each router. Unsurprisingly, the difference between the maximum latency observed during the congestion attack and the baseline measurement is significantly higher when attacking the correct first hop compared to attacking other routers. Also included for comparison is the average max latency over all iterations (also higher for the correct first hop), and the number of runs.

uring all Tor relays to refuse incoming connections from known exit relays, but even this approach does not entirely solve the problem: the attacker can use any external proxies he likes (e.g. open proxies, unlisted Tor relays, other anonymity networks) to “glue” his circuits together. Assuming external proxies with sufficient aggregate bandwidth are available for gluing, he can build a chain of circuits with arbitrary length. Note that the solution proposed in [30] — limiting circuit construction to trees — does not address this issue; furthermore, it increases overheads and implementation complexity far beyond the change proposed here and (contrary to the claims in [30]) may also have an impact on anonymity, since it requires Tor to fundamentally change the way circuits are constructed. We leave a full solution to this problem as an open research question.

Finally, given that strong adversaries may be able to mount latency altering attacks without Tor’s “help”, Tor users might consider using a longer path length than the minimalistic default of three. This would involve changes to Tor, as currently the only way for a user to change the default path length would be to edit and re-compile the code (probably out of scope for a “normal” user). While the presented attack can be made to work for longer paths, the number of false positives and the time required for a successful path discovery increase significantly with each extra hop. Using a random path length between four and six would furthermore require the adversary to confirm that the first hop was actually found (by determining that none of the other Tor routers could be a predecessor). Naturally, increasing the path length from three to six would significantly increase the latency and bandwidth requirements of the Tor network and might also hurt with respect to other attacks [2].

6 Low-cost Traffic Analysis Failure Against Modern Tor

We attempted to reproduce Murdoch and Danezis’s work [27] on the Tor network of 2008. Murdoch provided us with their code and statistical analysis framework which performs their congestion attack while measuring the latency of the circuit. Their analysis also determines the average latency and uses normalized latencies as the strength of the signal.

The main difference in terms of how data is obtained between Murdoch and Danezis and the attack presented in Section 3 is that Murdoch and Danezis use a circuit constructed by the attacker to measure the latency introduced by the victim circuit whereas our attack uses a circuit constructed by the victim to measure the latency introduced by the attacker.

As in this paper, the adversary implemented by Murdoch and Danezis repeatedly switches the congestion attack on and off; a high correlation between the presence of high latency values and the congestion attack being active is used to determine that a particular router is on the circuit. If such a correlation is absent for the correct router, the attack produces false negatives and fails. If a strong correlation is present between high latency values and random time periods (without an active attack) then the attack produces false positives and also fails.

Fig. 10 shows examples of our attempts at the method used in [27], two with the congestion attack being active and two without. Our experiments reproduced Murdoch and Danezis’s attack setup where the attacker tries to measure the congestion caused by the victim’s circuit. Note that in the graphs on the right, the congestion attack was run against a Tor router unrelated to the circuit

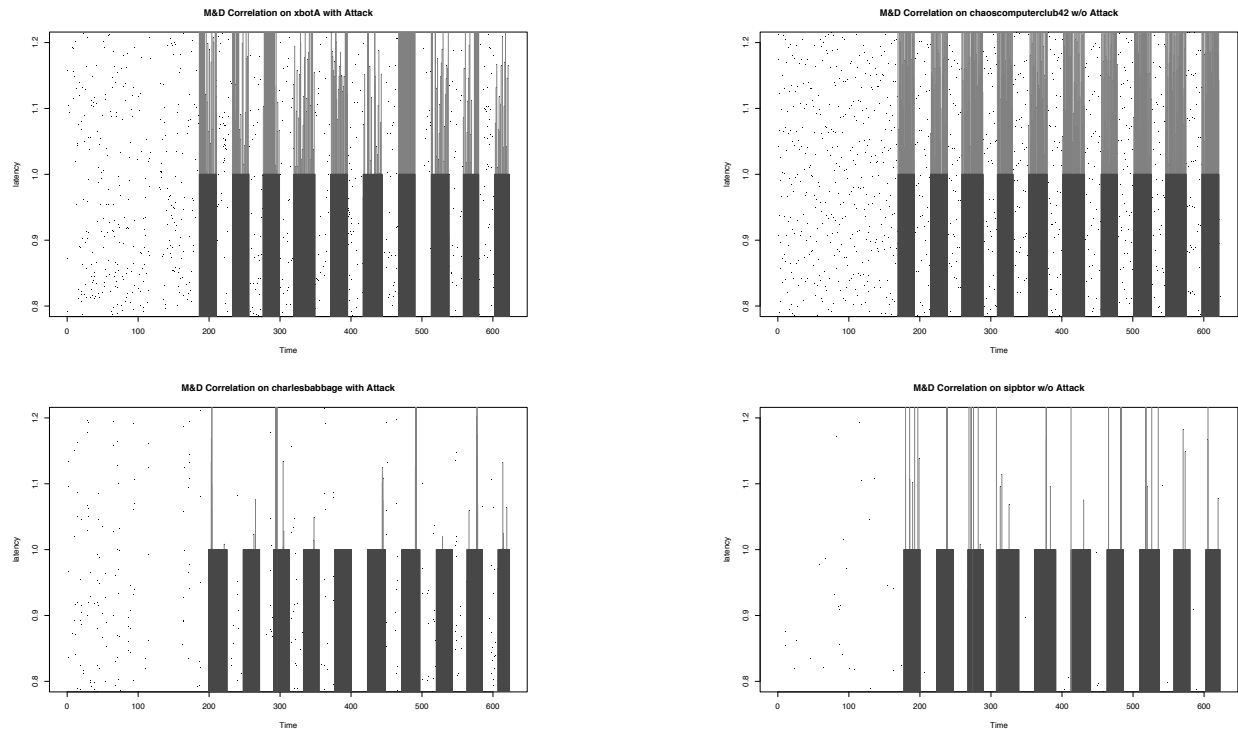


Figure 10: These graphs show four runs of the method used in [27], two with the congestion attack being active (on the left) and two without (on the right). The figure plots the observed latency of a router over time. Blue bars are used to indicate when the congestion attack was active; in the case of the graphs on the right the attack was active on an unrelated circuit. Red lines are drawn to latency values above average to mark latencies that correlate with the attack, according to the Murdoch and Danezis style analysis.

and thus inactive for the circuit that was measured. Any correlation observed in this case implies that Murdoch and Danezis’s attack produces false positives. The “visual” look of the graphs is the same whether the attack is targeted at that relay or not. Specifically, the graphs on the right suggest a similar correlation pattern even when the attack was “off” (or targeting unrelated Tor routers). This is due to the high volume of traffic on today’s Tor network causing baseline congestion which makes their analysis too indiscriminate.

Table 3 shows some representative correlation values that were computed using the statistical analysis from [27] when performed on the modern Tor network. Note that the correlation values are high regardless of whether or not the congestion attack was actually performed on the respective router. For Murdoch and Danezis’s analysis to work, high correlation values should only appear for the attacked router.

The problem with Murdoch and Danezis’s attack and analysis is not primarily with the statistical method; the single-circuit attack itself is simply not generating a sufficiently strong signal on the modern network. Fig. 11

plots the baseline latencies of Tor routers as well as the latencies of routers subjected to Murdoch and Danezis’s congestion attack in the style we used in Fig. 6. There are hardly any noticeable differences between routers under Murdoch and Danezis’s congestion attack and the baseline. Fig. 12 shows the latency histograms for the same data; in contrast to the histograms in Fig. 6 there is little difference between the histograms for the baseline and the attack.

In conclusion, due to the large amount of traffic on the modern Tor network, Murdoch and Danezis’s analysis is unable to differentiate between normal congestion and congestion caused by the attacker; the small amount of congestion caused by Murdoch and Danezis is lost in the noise of the network. As a result, their analysis produces many false positives and false negatives. While these experiments only represent a limited case-study and while Murdoch and Danezis’s analysis may still work in some cases, we never got reasonable results on the modern Tor network.

Router	Correlation	Attacked?	Peak BW	Configured BW
morphismherrex	1.43	Yes	222 kB/s	201 kB/s
chaoscomputerclub23	1.34	No	5414 kB/s	5120 kB/s
humanistischeunion1	1.18	No	5195 kB/s	6000 kB/s
mikezhangwithtor	1.07	No	1848 kB/s	2000 kB/s
hummingbird	1.03	No	710 kB/s	600 kB/s
chaoscomputerclub42	1.00	Yes	1704 kB/s	5120 kB/s
degaussYourself	1.00	No	4013 kB/s	4096 kB/s
ephemera	0.91	Yes	445 kB/s	150 kB/s
fissefjaes	0.99	Yes	382 kB/s	50 kB/s
zymurgy	0.86	Yes	230 kB/s	100 kB/s
charlesbabbage	0.53	Yes	2604 kB/s	1300 kB/s

Table 3: This table shows the correlation values calculated using the Murdoch and Danezis’s attack on the Tor network in Spring of 2008. False positives and false negatives are both abundant; many naturally congested routers show a strong correlation suggesting they are part of the circuit when they are not.

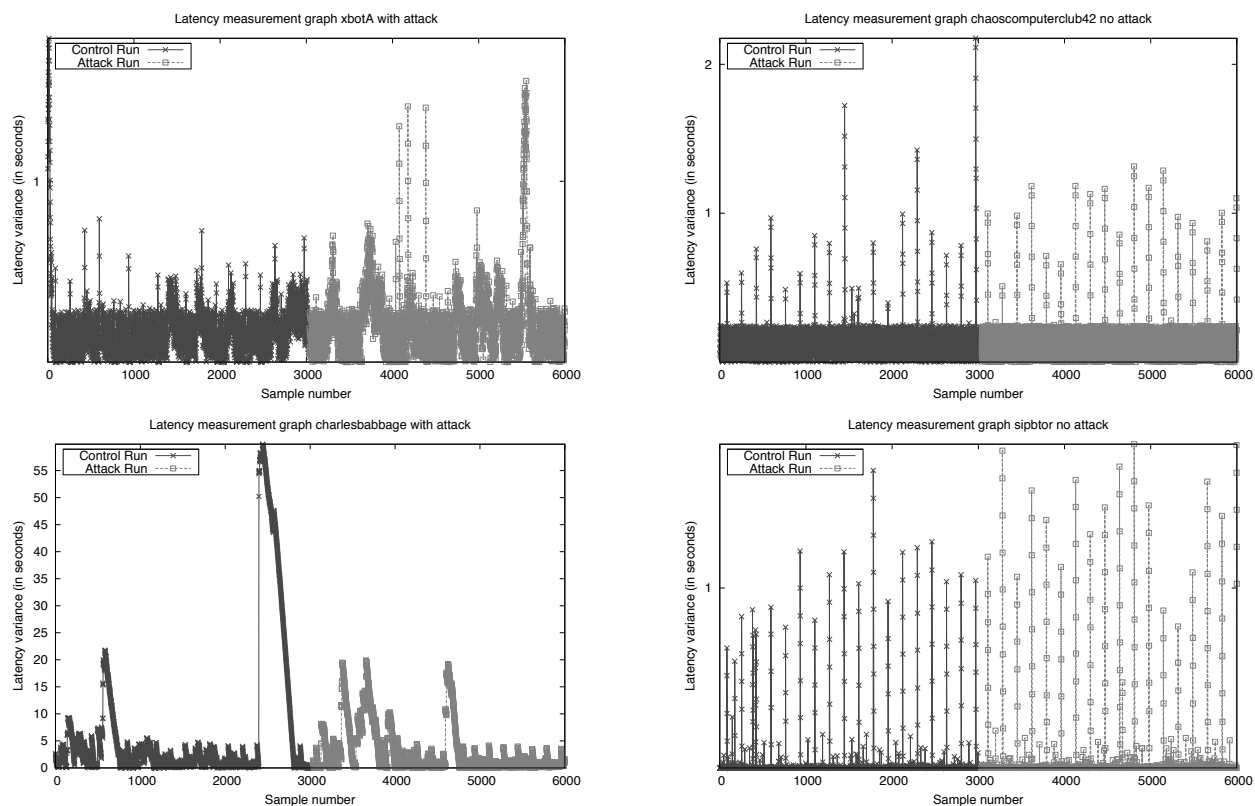


Figure 11: These graphs correspond to Fig. 10, showing the same attack in the style we used in Fig. 5. Note that during the attack phase the congestion circuit is turned on and off just as illustrated in Fig. 10. For all four routers the latency measurements are almost identical whether the attack was present or not.

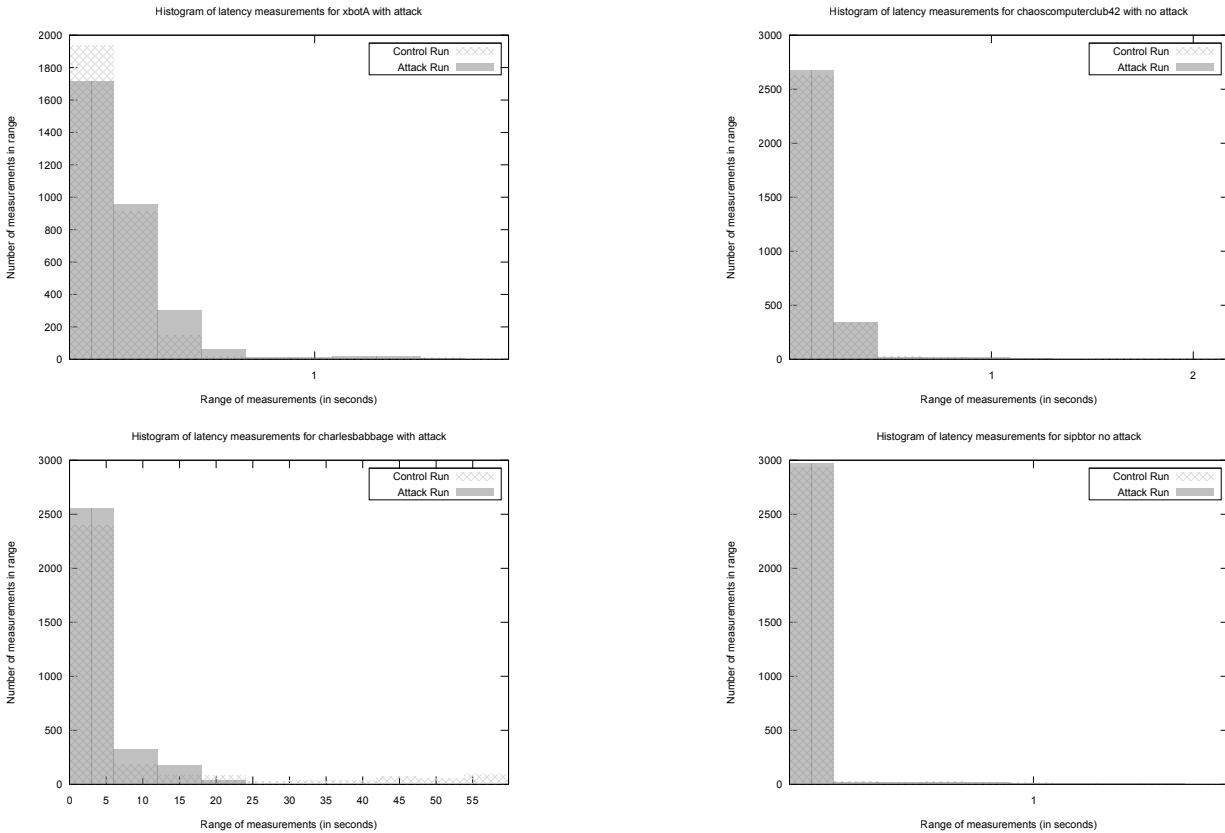


Figure 12: Once more we show the same data for comparison as shown in Fig. 10, this time in the histogram style we use in Fig. 6. The overlap between the control run and the attack run is difficult to see due to the similarity of latency distributions.

7 Conclusion

The possibility of constructing circuits of arbitrary length was previously seen as a minor problem that could lead to a DoS attack on Tor. This work shows that the problem is more serious, in that an adversary could use such circuits to improve methods for determining the path that packets take through the Tor network. Furthermore, Tor’s minimalistic default choice to use circuits of length three is questionable, given that an adversary controlling an exit node would only need to recover a tiny amount of information to learn the entire circuit. We have made some minimal changes to the Tor protocol that make it more difficult (but not impossible) for an adversary to construct long circuits.

Acknowledgments

This research was supported in part by the NLnet Foundation from the Netherlands (<http://nlnet.nl/>) and under NSF Grant No. 0416969. The authors thank

P. Eckersley for finding a problem in an earlier draft of the paper and K. Grothoff for editing.

References

- [1] BACK, A., MÖLLER, U., AND STIGLIC, A. Traffic analysis attacks and trade-offs in anonymity providing systems. In *Proceedings of Information Hiding Workshop (IH 2001)* (April 2001), I. S. Moskowitz, Ed., Springer-Verlag, LNCS 2137, pp. 245–257.
- [2] BORISOV, N., DANEZIS, G., MITTAL, P., AND TABRIZ, P. Denial of service or denial of security? How attacks on reliability can compromise anonymity. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security* (New York, NY, USA, October 2007), ACM, pp. 92–102.
- [3] CHAUM, D. L. Untraceable electronic mail, return addresses, and digital pseudonyms. *Commun. ACM* 24, 2 (February 1981), 84–90.
- [4] DAI, W. Two attacks against freedom. <http://www.weidai.com/freedom-attacks.txt>, 2000.
- [5] DANEZIS, G., DINGLEDINE, R., AND MATHEWSON, N. Mixminion: Design of a Type III Anonymous Remailer Protocol. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy* (May 2003), pp. 2–15.

- [6] DANIEL STENBERG, E. A. libcurl, 1998–2009. Open Source C-based multi-platform file transfer library.
- [7] DESMEDT, Y., AND KUROSAWA, K. How to break a practical MIX and design a new one. In *Advances in Cryptology — Eurocrypt 2000, Proceedings* (2000), Springer-Verlag, LNCS 1807, pp. 557–572.
- [8] DIAZ, C., AND SERJANTOV, A. Generalising mixes. In *Proceedings of Privacy Enhancing Technologies workshop (PET 2003)* (March 2003), R. Dingledine, Ed., Springer-Verlag, LNCS 2760, pp. 18–31.
- [9] DINGLEDINE, R. Tor proposal 110: Avoiding infinite length circuits. <https://svn.torproject.org/svn/tor/trunk/doc/spec/proposals/110-avoid-infinite-circuits.txt>, March 2007.
- [10] DINGLEDINE, R. Tor bridges specification. Tech. rep., The Tor Project, <https://svn.torproject.org/svn/tor/trunk/doc/spec/bridges-spec.txt>, 2008.
- [11] DINGLEDINE, R., AND MATHEWSON, N. Design of a blocking-resistant anonymity system. Tech. rep., The Tor Project, <https://svn.torproject.org/svn/tor/trunk/doc/design-paper/blocking.pdf>, 2007.
- [12] DINGLEDINE, R., MATHEWSON, N., AND SYVERSON, P. Tor: The second-generation onion router. In *Proceedings of the 13th USENIX Security Symposium* (August 2004).
- [13] FIELDING, R., GETTYS, J., MOGUL, J., FRYSTYK, H., MASINTER, L., LEACH, P., AND BERNERS-LEE, T. *RFC 2616: Hypertext Transfer Protocol — HTTP/1.1*. The Internet Society, June 1999.
- [14] FREEDMAN, M. J., AND MORRIS, R. Tarzan: a peer-to-peer anonymizing network layer. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security* (New York, NY, USA, November 2002), ACM, pp. 193–206.
- [15] FREEDMAN, M. J., SIT, E., CATES, J., AND MORRIS, R. Introducing tarzan, a peer-to-peer anonymizing network layer. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems* (London, UK, 2002), Springer-Verlag, pp. 121–129.
- [16] GOLDSCHLAG, D. M., REED, M. G., AND SYVERSON, P. F. Hiding Routing Information. In *Proceedings of Information Hiding: First International Workshop* (May 1996), R. Anderson, Ed., Springer-Verlag, LNCS 1174, pp. 137–150.
- [17] GÜLCÜ, C., AND TSUDIK, G. Mixing E-mail with Babel. In *Proceedings of the Network and Distributed Security Symposium - NDSS '96* (February 1996), IEEE, pp. 2–16.
- [18] HAN, J., AND LIU, Y. Rumor riding: Anonymizing unstructured peer-to-peer systems. In *ICNP '06: Proceedings of the Proceedings of the 2006 IEEE International Conference on Network Protocols* (Washington, DC, USA, Nov 2006), IEEE Computer Society, pp. 22–31.
- [19] HOPPER, N., VASSERMAN, E. Y., AND CHAN-TIN, E. How much anonymity does network latency leak? In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security* (New York, NY, USA, October 2007), ACM, pp. 82–91.
- [20] KEIL, F., SCHMIDT, D., ET AL. Privoxy - a privacy enhancing web proxy. <http://www.privoxy.org/>.
- [21] KESDOGAN, D., EGNER, J., AND BÜSCHKES, R. Stop-and-go MIXes: Providing probabilistic anonymity in an open system. In *Proceedings of the Second International Workshop on Information Hiding* (London, UK, 1998), Springer-Verlag, LNCS 1525, pp. 83–98.
- [22] LANDSIEDEL, O., PIMENIDIS, A., WEHRLE, K., NIEDERMAYER, H., AND CARLE, G. Dynamic multipath onion routing in anonymous peer-to-peer overlay networks. *Global Telecommunications Conference, 2007. GLOBECOM '07. IEEE* (Nov. 2007), 64–69.
- [23] LEVINE, B. N., REITER, M. K., WANG, C., AND WRIGHT, M. K. Timing attacks in low-latency mix-based systems. In *Proceedings of Financial Cryptography (FC '04)* (February 2004), A. Juels, Ed., Springer-Verlag, LNCS 3110, pp. 251–265.
- [24] MCLACHLAN, J., AND HOPPER, N. Don't clog the queue! circuit clogging and mitigation in p2p anonymity schemes. In *Financial Cryptography* (2008), G. Tsudik, Ed., vol. 5143 of *Lecture Notes in Computer Science*, Springer, pp. 31–46.
- [25] MÖLLER, U., COTTRELL, L., PALFRADER, P., AND SASAMAN, L. Mixmaster Protocol — Version 2. IETF Internet Draft, December 2004.
- [26] MURDOCH, S. J. *Covert channel vulnerabilities in anonymity systems*. PhD thesis, University of Cambridge, December 2007.
- [27] MURDOCH, S. J., AND DANEZIS, G. Low-cost traffic analysis of Tor. In *SP '05: Proceedings of the 2005 IEEE Symposium on Security and Privacy* (Washington, DC, USA, May 2005), IEEE Computer Society, pp. 183–195.
- [28] NAMBIAR, A., AND WRIGHT, M. Salsa: a structured approach to large-scale anonymity. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security* (New York, NY, USA, October 2006), ACM, pp. 17–26.
- [29] ØVERLIER, L., AND SYVERSON, P. Locating hidden servers. In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy* (Washington, DC, USA, May 2006), IEEE Computer Society, pp. 100–114.
- [30] PAPPAS, V., ATHANASOPOULOS, E., IOANNIDIS, S., AND MARKATOS, E. P. Compromising anonymity using packet spinning. In *Proceedings of the 11th Information Security Conference (ISC 2008)* (2008), T.-C. Wu, C.-L. Lei, V. Rijmen, and D.-T. Lee, Eds., vol. 5222 of *Lecture Notes in Computer Science*, Springer, pp. 161–174.
- [31] PERRY, M., AND SQUIRES, S. <https://www.torproject.org/torbutton/>, 2009.
- [32] PFITZMANN, A., PFITZMANN, B., AND Waidner, M. ISDN-mixes: Untraceable communication with very small bandwidth overhead. In *Proceedings of the GIITG Conference on Communication in Distributed Systems* (February 1991), pp. 451–463.
- [33] RENNARD, M., AND PLATTNER, B. Introducing MorphMix: Peer-to-Peer based Anonymous Internet Usage with Collusion Detection. In *WPES '02: Proceedings of the 2002 ACM workshop on Privacy in the Electronic Society* (New York, NY, USA, November 2002), ACM, pp. 91–102.
- [34] SERJANTOV, A., DINGLEDINE, R., AND SYVERSON, P. From a trickle to a flood: Active attacks on several mix types. In *IH '02: Revised Papers from the 5th International Workshop on Information Hiding* (London, UK, 2003), F. Petitcolas, Ed., Springer-Verlag, LNCS 2578, pp. 36–52.
- [35] SHMATIKOV, V., AND WANG, M.-H. Timing analysis in low-latency mix networks: Attacks and defenses. In *Proceedings of the 11th European Symposium on Research in Computer Security (ESORICS)* (September 2006), pp. 236–252.
- [36] WIANGSRIPANAWAN, R., SUSILO, W., AND SAFAVI-NAINI, R. Design principles for low latency anonymous network systems secure against timing attacks. In *Proceedings of the fifth Australasian symposium on ACSW frontiers (ACSW '07)* (Darlinghurst, Australia, Australia, 2007), Australian Computer Society, Inc, pp. 183–191.

Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors

Periklis Akritidis,^{*} Manuel Costa,[†] Miguel Castro,[†] Steven Hand^{*}

^{*}Computer Laboratory
University of Cambridge, UK
{pa280,smh22}@cl.cam.ac.uk

[†]Microsoft Research
Cambridge, UK
{manuelc,mcastro}@microsoft.com

Abstract

Attacks that exploit out-of-bounds errors in C and C++ programs are still prevalent despite many years of research on bounds checking. Previous backwards compatible bounds checking techniques, which can be applied to unmodified C and C++ programs, maintain a data structure with the bounds for each allocated object and perform lookups in this data structure to check if pointers remain within bounds. This data structure can grow large and the lookups are expensive.

In this paper we present a backwards compatible bounds checking technique that substantially reduces performance overhead. The key insight is to constrain the sizes of allocated memory regions and their alignment to enable efficient bounds lookups and hence efficient bounds checks at runtime. Our technique has low overhead in practice—only 8% throughput decrease for Apache—and is more than two times faster than the fastest previous technique and about five times faster—using less memory—than recording object bounds using a splay tree.

1 Introduction

Bounds checking C and C++ code protects against a wide range of common vulnerabilities. The challenge has been making bounds checking fast enough for production use and at the same time backwards compatible with binary libraries to allow incremental deployment. Solutions using *fat pointers* [24, 18] extend the pointer representation with bounds information. This enables efficient bounds checks but breaks backwards compatibility because increasing the pointer size changes the memory layout of data structures. Backwards compatible bounds checking techniques [19, 30, 36, 15] use a separate data structure to lookup bounds information. Initial attempts incurred a significant overhead [19, 30, 36] (typically 2x–10x) be-

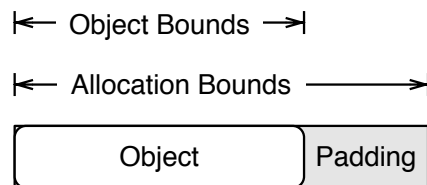


Figure 1: Allocated memory is often padded to a particular alignment boundary, and hence can be larger than the requested object size. By checking *allocation bounds* rather than *object bounds*, we allow benign accesses to the padding, but can significantly reduce the cost of bounds lookups at runtime.

cause looking up bounds is expensive and the data structure can grow large. More recent work [15] has applied sophisticated static pointer analysis to reduce the number of bounds lookups; this managed to reduce the runtime overhead on the Olden benchmarks to 12% on average.

In this paper we present *baggy bounds checking*, a backwards compatible bounds checking technique that reduces the cost of bounds checks. We achieve this by enforcing *allocation bounds* rather than precise object bounds, as shown in Figure 1. Since memory allocators pad object allocations to align the pointers they return, there is a class of benign out-of-bounds errors that violate the object bounds but fall within the allocation bounds. Previous work [4, 19, 2] has exploited this property in a variety of ways.

Here we apply it to efficient backwards compatible bounds checking. We use a binary buddy allocator to enable a compact representation of the allocation bounds: since all allocation sizes are powers of two, a single byte is sufficient to store the binary logarithm of the allocation

size. Furthermore, there is no need to store additional information because the base address of an allocation with size s can be computed by clearing the $\log_2(s)$ least significant bits of any pointer to the allocated region. This allows us to use a space and time efficient data structure for the bounds table. We use a contiguous array instead of a more expensive data structure (such as the splay trees used in previous work). It also provides us with an elegant way to deal with common cases of temporarily out-of-bounds pointers. We describe our design in more detail in Section 2.

We implemented *baggy bounds checking* as a compiler plug-in for the Microsoft Phoenix [22] code generation framework, along with additional run time components (Section 3). The plug-in inserts code to check bounds for all pointer arithmetic that cannot be statically proven safe, and to align and pad stack variables where necessary. The run time component includes a binary buddy allocator for heap allocations, and user-space virtual memory handlers for growing the bounds table on demand.

In Section 4 we evaluate the performance of our system using the Olden benchmark (to enable a direct comparison with Dhurjati and Adve [15]) and SPECINT 2000. We compare our space overhead with a version of our system that uses the splay tree implementation from [19, 30]. We also verify the efficacy of our system in preventing attacks using the test suite described in [34], and run a number of security critical COTS components to confirm its applicability.

Section 5 describes our design and implementation for 64-bit architectures. These architectures typically have “spare” bits within pointers, and we describe a scheme that uses these to encode bounds information directly in the pointer rather than using a separate lookup table. Our comparative evaluation shows that the performance benefit of using these spare bits to encode bounds may not in general justify the additional complexity; however using them just to encode information to recover the bounds for out-of-bounds pointers may be worthwhile.

Finally we survey related work (Section 6), discuss limitations and possible future work (Section 7) and conclude (Section 8).

2 Design

2.1 Baggy Bounds Checking

Our system shares the overall architecture of backwards compatible bounds checking systems for C/C++ (Fig-

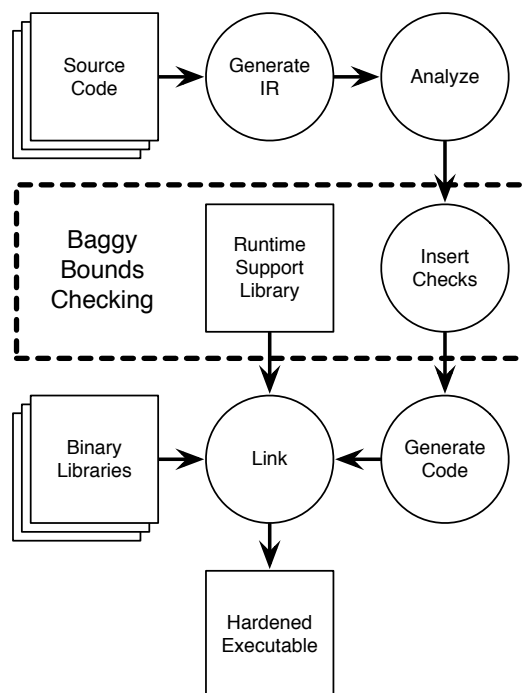


Figure 2: Overall system architecture, with our contribution highlighted within the dashed box.

ure 2). It converts source code to an intermediate representation (IR), finds potentially unsafe pointer arithmetic operations, and inserts checks to ensure their results are within bounds. Then, it links the generated code with our runtime library and binary libraries—compiled with or without checks—to create a hardened executable.

We use the *referent object* approach for bounds checking introduced by Jones and Kelly [19]. Given an in-bounds pointer to an object, this approach ensures that any derived pointer points to the same object. It records bounds information for each object in a *bounds table*. This table is updated on allocation and deallocation of objects: this is done by the `malloc` family of functions for heap-based objects; on function entry and exit for stack-based objects; and on program startup for global objects.

The referent object approach performs bounds checks on pointer arithmetic. It uses the source pointer to lookup the bounds in the table, performs the operation, and checks if the destination pointer remains in bounds. If the destination pointer does not point to the same object, we mark it out-of-bounds to prevent any dereference (as in [30, 15]). However we permit its use in further pointer arithmetic, since it may ultimately result in an in-bounds pointer. The marking mechanism is described in detail in Section 2.4.

Baggy bounds checking uses a very compact repre-

sentation for bounds information. Previous techniques recorded a pointer to the start of the object and its size in the bounds table, which requires at least eight bytes. We pad and align objects to powers of two and enforce allocation bounds instead of object bounds. This enables us to use a single byte to encode bounds information. We store the binary logarithm of the allocation size in the bounds table:

```
e = log2(size);
```

Given this information, we can recover the allocation size and a pointer to the start of the allocation with:

```
size = 1 << e;
```

```
base = p & ~(size-1);
```

To convert from an in-bounds pointer to the bounds for the object we require a *bounds table*. Previous solutions based on the referent object approach (such as [19, 30, 15]) have implemented the bounds table using a splay tree.

Baggy bounds, by contrast, implement the bounds table using a contiguous array. The table is small because each entry uses a single byte. Additionally, we partition memory into aligned *slots* with *slot_size* bytes. The bounds table has an entry for each slot rather than an entry per byte. So the space overhead of the table is $1/\text{slot_size}$, and we can tune *slot_size* to balance memory waste between padding and table size. We align objects to slot boundaries to ensure that no two objects share a slot.

Accesses to the table are fast. To obtain a pointer to the entry corresponding to an address, we right-shift the address by the constant $\log_2(\text{slot_size})$ and add the constant table base. We can use this pointer to retrieve the bounds information with a single memory access, instead of having to traverse and splay a splay tree (as in previous solutions).

Note that baggy bounds checking permits benign out-of-bounds accesses to the memory padding after an object. This does not compromise security because these accesses cannot write or read other objects. They cannot be exploited for typical attacks such as (a) overwriting a return address, function pointer or other security critical data; or (b) reading sensitive information from another object, such as a password.

We also defend against a less obvious attack where the program reads values from the padding area that were originally written to a deleted object that occupied the same memory. We prevent this attack by clearing the padding on memory allocation.

Pointer arithmetic operation:

```
p' = p + i
```

Explicit bounds check:

```
size = 1 << table[p>>slot_size]
base = p & ~(size-1)
```

```
p' >= base && p' - base < size
```

Optimized bounds check:

```
(p^p') >> table[p>>slot_size] == 0
```

Figure 3: Baggy bounds enables optimized bounds checks: we can verify that pointer p' derived from pointer p is within bounds by simply checking that p and p' have the same prefix with only the e least significant bits modified, where e is the binary logarithm of the allocation size.

2.2 Efficient Checks

In general, bounds checking the result p' of pointer arithmetic on p involves two comparisons: one against the lower bound and one against the upper bound, as shown in Figure 3.

We devised an optimized bounds check that does not even need to compute the lower and upper bounds. It uses the value of p and the value of the binary logarithm of the allocation size, e , retrieved from the bounds table. The constraints on allocation size and alignment ensure that p' is within the allocation bounds if it differs from p only in the e least significant bits. Therefore, it is sufficient to shift $p \wedge p'$ by e and check if the result is zero, as shown in Figure 3.

Furthermore, for pointers p' where $\text{sizeof}(*p') > 1$, we also need to check that $(\text{char} *) p' + \text{sizeof}(*p') - 1$ is within bounds to prevent a subsequent access to $*p'$ from crossing the allocation bounds. Baggy bounds checking can avoid this extra check if p' points to a built-in type. Aligned accesses to these types cannot overlap an allocation boundary because their size is a power of two and is less than *slot_size*. When checking pointers to structures that do not satisfy these constraints, we perform both checks.

2.3 Interoperability

Baggy bounds checking works even when instrumented code is linked against libraries that are not instrumented.

The library code works without change because it performs no checks but it is necessary to ensure that instrumented code works when accessing memory allocated in an uninstrumented library. This form of interoperability is important because some libraries are distributed in binary form.

We achieve interoperability by using the binary logarithm of the maximum allocation size as the default value for bounds table entries. Instrumented code overwrites the default value on allocations with the logarithm of the allocation size and restores the default value on deallocations. This ensures that table entries for objects allocated in uninstrumented libraries inherit the default value. Therefore, instrumented code can perform checks as normal when accessing memory allocated in a library, but checking is effectively disabled for these accesses. We could intercept heap allocations in library code at link time and use the buddy allocator to enable bounds checks on accesses to library-allocated memory, but this is not done in the current prototype.

2.4 Support for Out-Of-Bounds Pointers

A pointer may legally point outside the object bounds in C. Such pointers should not be dereferenced but can be compared and used in pointer arithmetic that can eventually result in a valid pointer that may be dereferenced by the program.

Out-of-bounds pointers present a challenge for the referent object approach because it relies on an in-bounds pointer to retrieve the object bounds. The C standard only allows out-of-bounds pointers to one element past the end of an array. Jones and Kelly [19] support these legal out-of-bounds pointers by padding objects with one byte. We did not use this technique because it interacts poorly with our constraints on allocation sizes: adding one byte to an allocation can double the allocated size in the common case where the requested allocation size is a power of two.

Many programs violate the C standard and generate illegal but harmless out-of-bounds pointers that they never dereference. Examples include faking a base one array by decrementing the pointer returned by `malloc` and other equally tasteless uses. CRED [30] improved on the Jones and Kelly bounds checker [19] by tracking such pointers using another auxiliary data structure. We did not use this approach because it adds overhead on deallocations of heap and local objects: when an object is deallocated the auxiliary data structure must be searched to remove entries tracking out-of-bounds pointers to the object. Additionally, entries in this auxiliary data struc-

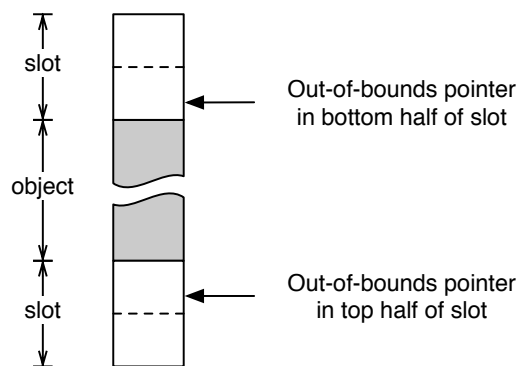


Figure 4: We can tell whether a pointer that is out-of-bounds by less than $slot_size/2$ is below or above an allocation. This lets us correctly adjust it to get a pointer to the object by respectively adding or subtracting $slot_size$.

ture may accumulate until their referent object is deallocated.

We handle out-of-bounds pointers within $slot_size/2$ bytes from the original object as follows. First, we mark out-of-bounds pointers to prevent them from being dereferenced (as in [15]). We use the memory protection hardware to prevent dereferences by setting the most significant bit in these pointers and by restricting the program to the lower half of the address space (this is often already the case for user-space programs). We can recover the original pointer by clearing the bit.

The next challenge is to recover a pointer to the referent object from the out-of-bounds pointer without resorting to an additional data structure. We can do this for the common case when out-of-bounds pointers are at most $slot_size/2$ bytes before or after the allocation. Since the allocation bounds are aligned to slot boundaries, we can find if a marked pointer is below or above the allocation by checking whether it lies in the top or bottom half of a memory slot respectively, as illustrated in Figure 4. We can recover a pointer to the referent object by adding or subtracting $slot_size$ bytes. This technique cannot handle pointers that go more than $slot_size/2$ bytes outside the original object. In Section 5.2, we show how to take advantage of the spare bits in pointers on 64 bit architectures to increase this range, and in Section 7 we discuss how we could add support for arbitrary out-of-bounds pointers while avoiding some of the problems of previous solutions.

It is not necessary to instrument pointer dereferences. Similarly, there is no need to instrument pointer equality comparisons because the comparison will be correct whether the pointers are out-of-bounds or not. But we need to instrument inequality comparisons to support

comparing an out-of-bounds pointer with an in-bounds one: the instrumentation must clear the high-order bit of the pointers before comparing them. We also instrument pointer differences in the same way.

Like previous bounds checking solutions [19, 30, 15], we do not support passing an out-of-bounds pointer to uninstrumented code. However, this case is rare. Previous work [30] did not encounter this case in several million lines of code.

2.5 Static Analysis

Bounds checking has relied heavily on static analysis to optimize performance [15]. Checks can be eliminated if it can be statically determined that a pointer is safe, i.e. always within bounds, or that a check is redundant due to a previous check. Furthermore, checks or just the bounds lookup can be hoisted out of loops. We have not implemented a sophisticated analysis and, instead, focused on making checks efficient.

Nevertheless, our prototype implements a simple intra-procedural analysis to detect safe pointer operations. We track allocation sizes and use the compiler's variable range analysis to eliminate checks that are statically shown to be within bounds. We also investigate an approach to hoist checks out of loops that is described in Section 3.

We also use static analysis to reduce the number of local variables that are padded and aligned. We only pad and align local variables that are indexed unsafely within the function, or whose address is taken, and therefore possibly leaked from the function. We call these variables *unsafe*.

3 Implementation

We used the Microsoft Phoenix [22] code generation framework to implement a prototype system for x86 machines running Microsoft Windows. The system consists of a plug-in to the Phoenix compiler and a runtime support library. In the rest of this section, we describe some implementation details.

3.1 Bounds Table

We chose a *slot_size* of 16 bytes to avoid penalizing small allocations. Therefore, we reserve $1/16^{th}$ of the address space for the bounds table. Since pages are allocated to the table on demand, this increases memory

utilization by only 6.25%. We reserve the address space required for the bounds table on program startup and install a user space page fault handler to allocate missing table pages on demand. All the bytes in these pages are initialized by the handler to the value 31, which encompasses all the addressable memory in the x86 (an allocation size of 2^{31} at base address 0). This prevents out-of-bounds errors when instrumented code accesses memory allocated by uninstrumented code.

3.2 Padding and Aligning

We use a binary buddy allocator to satisfy the size and alignment constraints on heap allocations. Binary buddy allocators provide low external fragmentation but suffer from internal fragmentation because they round allocation sizes to powers of two. This shortcoming is put to good use in our system. Our buddy allocator implementation supports a minimum allocation size of 16 bytes, which matches our *slot_size* parameter, to ensure that no two objects share the same slot.

We instrument the program to use our version of `malloc`-style heap allocation functions based on the buddy allocator. These functions set the corresponding bounds table entries and zero the padding area after an object. For local variables, we align the stack frames of functions that contain unsafe local variables at runtime and we instrument the function entry to zero the padding and update the appropriate bounds table entries. We also instrument function exit to reset table entries to 31 for interoperability when uninstrumented code reuses stack memory. We align and pad static variables at compile time and their bounds table entries are initialized when the program starts up.

Unsafe function arguments are problematic because padding and aligning them would violate the calling convention. Instead, we copy them on function entry to appropriately aligned and padded local variables and we change all references to use the copies (except for uses of `va_list` that need the address of the last explicit argument to correctly extract subsequent arguments). This preserves the calling convention while enabling bounds checking for function arguments.

The Windows runtime cannot align stack objects to more than 8K nor static objects to more than 4K (configurable using the `/ALIGN` linker switch). We could replace these large stack and static allocations with heap allocations to remove this limitation but our current prototype sets the bounds table entries for these objects to 31.

Zeroing the padding after an object can increase space and time overhead for large padding areas. We avoid this

overhead by relying on the operating system to zero allocated pages on demand. Then we track the subset of these pages that is modified and we zero padding areas in these pages on allocations. Similar issues are discussed in [9] and the standard allocator uses a similar technique for `calloc`. Our buddy allocator also uses this technique to avoid explicitly zeroing large memory areas allocated with `calloc`.

3.3 Checks

We add checks for each pointer arithmetic and array indexing operation but, following [15], we do not instrument accesses to scalar fields in structures and we do not check pointer dereferences. This facilitates a direct comparison with [15]. We could easily modify our implementation to perform these checks, for example, using the technique described in [14].

We optimize bounds checks for the common case of in-bounds pointers. To avoid checking if a pointer is marked out-of-bounds in the fast path, we set all the entries in the bounds table that correspond to out-of-bounds pointers to zero. Since out-of-bounds pointers have their most significant bit set, we implement this by mapping all the virtual memory pages in the top half of the bounds table to a shared zero page. This ensures that our slow path handler is invoked on any arithmetic operation involving a pointer marked out-of-bounds.

bounds lookup	{	mov eax, buf shr eax, 4 mov al, byte ptr [TABLE+eax]
pointer arithmetic	{	char *p = buf[i];
bounds check	{	mov ebx, buf xor ebx, p shr ebx, al jz ok p = slowPath(buf, p) ok:

Figure 5: Code sequence inserted to check unsafe pointer arithmetic.

Figure 5 shows the x86 code sequence that we insert before an example pointer arithmetic operation. First, the source pointer, `buf`, is right shifted to obtain the index of the bounds table entry for the corresponding slot. Then the logarithm of the allocation size e is loaded from the bounds table into register `al`. The result of the pointer arithmetic, `p`, is xored with the source pointer, `buf`, and right shifted by `al` to discard the bottom bits. If `buf` and `p` are both within the allocation bounds they can only

differ in the $\log_2 e$ least significant bits (as discussed before). So if the zero flag is set, `p` is within the allocation bounds. Otherwise, the `slowPath` function is called.

The `slowPath` function starts by checking if `buf` has been marked out-of-bounds. In this case, it obtains the referent object as described in 2.4, resets the most significant bit in `p`, and returns the result if it is within bounds. Otherwise, the result is out-of-bounds. If the result is out-of-bounds by more than half a slot, the function signals an error. Otherwise, it marks the result out-of-bounds and returns it. Any attempt to dereference the returned pointer will trigger an exception. To avoid disturbing register allocation in the fast path, the `slowPath` function uses a special calling convention that saves and restores all registers.

As discussed in Section 3.3, we must add `sizeof(*p)` to the result and perform a second check if the pointer is not a pointer to a built-in type. In this case, `buf` is a `char*`.

Similar to previous work, we provide bounds checking wrappers for Standard C Library functions such as `strcpy` and `memcpy` that operate on pointers. We replace during instrumentation calls to these functions with calls to their wrappers.

3.4 Optimizations

Typical optimizations used with bounds checking include eliminating redundant checks, hoisting checks out of loops, or hoisting just bounds table lookups out of loops. Optimization of inner loops can have a dramatic impact on performance. We experimented with hoisting bounds table lookups out of loops when all accesses inside a loop body are to the same object. Unfortunately, performance did not improve significantly, probably because our bounds lookups are inexpensive and hoisting can adversely effect register allocation.

Hoisting the whole check out of a loop is preferable when static analysis can determine symbolic bounds on the pointer values in the loop body. However, hoisting out the check is only possible if the analysis can determine that these bounds are guaranteed to be reached in every execution. Figure 6 shows an example where the loop bounds are easy to determine but the loop may terminate before reaching the upper bound. Hoisting out the check would trigger a false alarm in runs where the loop exits before violating the bounds.

We experimented with an approach that generates two versions of the loop code, one with checks and one without. We switch between the two versions on loop entry.

In the example of Figure 6, we lookup the bounds of p and if n does not exceed the size we run the unchecked version of the loop. Otherwise, we run the checked version.

```

for (i = 0; i < n; i++) {
    if (p[i] == 0) break;
    ASSERT(IN_BOUNDS(p, &p[i]));
    p[i] = 0;
}

↓

if (IN_BOUNDS(p, &p[n-1])) {
    for (i = 0; i < n; i++) {
        if (p[i] == 0) break;
        p[i] = 0;
    }
} else {
    for (i = 0; i < n; i++) {
        if (p[i] == 0) break;
        ASSERT(IN_BOUNDS(p, &p[i]));
        p[i] = 0;
    }
}

```

Figure 6: The compiler’s range analysis can determine that the range of variable i is at most $0 \dots n-1$. However, the loop may exit before i reaches $n-1$. To prevent erroneously raising an error, we fall back to an instrumented version of the loop if the hoisted check fails.

4 Experimental Evaluation

In this section we evaluate the performance of our system using CPU intensive benchmarks, its effectiveness in preventing attacks using a buffer overflow suite, and its usability by building and measuring the performance of real world security critical code.

4.1 Performance

We evaluate the time and peak memory overhead of our system using the Olden benchmarks and SPECINT 2000. We chose these benchmarks in part to allow a comparison against results reported for some other solutions [15, 36, 23]. In addition, to enable a more detailed comparison with splay-tree-based approaches—including measuring their space overhead—we implemented a variant of our approach which uses the splay tree code from previous systems [19, 30]. This implementation uses the standard allocator and is lacking support for illegal out-of-bounds pointers, but is otherwise identical to our system. We compiled all benchmarks with the Phoenix compiler using `/O2` optimization level

and ran them on a 2.33 GHz Intel Core 2 Duo processor with 2 GB of RAM.

From SPECINT 2000 we excluded `eon` since it uses C++ which we do not yet support. For our splay-tree-based implementation only we did not run `vpr` due to its lack of support for illegal out-of-bounds pointers. We also could not run `gcc` because of code that subtracted a pointer from a NULL pointer and subtracted the result from NULL again to recover the pointer. Running this would require more comprehensive support for out-of-bounds pointers (such as that described in [30], as we propose in Section 7).

We made the following modifications to some of the benchmarks: First, we modified `parser` from SPECINT 2000 to fix an overflow that triggered a bound error when using the splay tree. It did not trigger an error with baggy bounds checking because in our runs the overflow was entirely contained in the allocation, but should it overlap another object during a run, the baggy checking would detect it. The unchecked program also survived our runs because the object was small enough for the overflow to be contained even in the padding added by the standard allocator.

Then, we had to modify `perlbnk` by changing two lines to prevent an out-of-bounds arithmetic whose result is never used and `gap` by changing 5 lines to avoid an out-of-bounds pointer. Both cases can be handled by the extension described in Section 5, but are not covered by the small out-of-bounds range supported by our 32-bit implementation and the splay-tree-based implementation.

Finally, we modified `mst` from Olden to disable a custom allocator that allocates 32 Kbyte chunks of memory at a time that are then broken down to 12 byte objects. This increases protection at the cost of memory allocation overhead and removes an unfair advantage for the splay tree whose time and space overheads are minimized when the tree contains just a few nodes, as well as baggy space overhead that benefits from the power of two allocation. This issue, shared with other systems offering protection at the memory block level [19, 30, 36, 15, 2], illustrates a frequent situation in C programs that may require tweaking memory allocation routines in the source code to take full advantage of checking. In this case merely changing a macro definition was sufficient.

We first ran the benchmarks replacing the standard allocator with our buddy system allocator to isolate its effects on performance, and then we ran them using our full system. For the Olden benchmarks, Figure 7 shows the execution time and Figure 8 the peak memory usage.

In Figure 7 we observe that some benchmarks in the Olden suite (`mst`, `health`) run significantly faster with

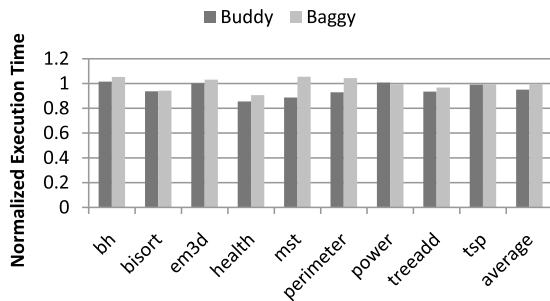


Figure 7: Execution time for the Olden benchmarks using the buddy allocator and our full system, normalized by the execution time using the standard system allocator without instrumentation.

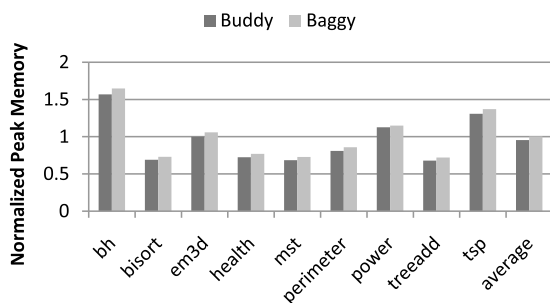


Figure 8: Peak memory use with the buddy allocator alone and with the full system for the Olden benchmarks, normalized by peak memory using the standard allocator without instrumentation.

the buddy allocator than with the standard one. These benchmarks are memory intensive and any memory savings reflect on the running time. In Figure 8 we can see that the buddy system uses less memory for these than the standard allocator. This is because these benchmarks contain numerous small allocations for which the padding to satisfy alignment requirements and the per-allocation metadata used by the standard allocator exceed the internal fragmentation of the buddy system.

This means that the average time overhead of the full system across the entire Olden suite is actually zero, because the positive effects of using the buddy allocator mask the costs of checks. The time overhead of the checks alone as measured against the buddy allocator as a baseline is 6%. The overhead of the fastest previous bounds checking system [15] on the same benchmarks and same protection (modulo allocation vs. object bounds) is 12%, but their system also benefits from the technique of pool allocation which can also be used independently. Based on the breakdown of results reported in [15], their overhead measured against the pool allocation is 15%, and it seems more reasonable to compare these two numbers,

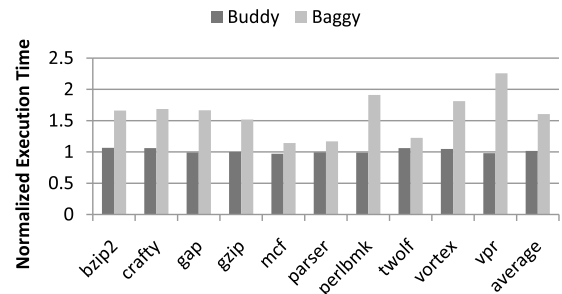


Figure 9: Execution time for SPECINT 2000 benchmarks using the buddy allocator and our full system, normalized by the execution time using the standard system allocator without instrumentation.

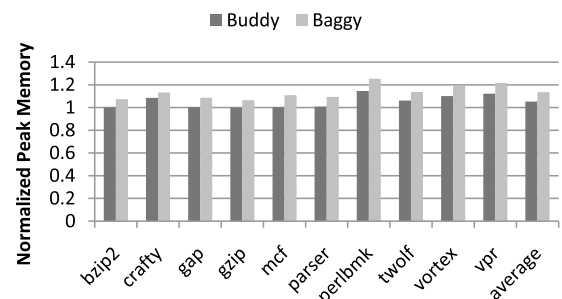


Figure 10: Peak memory use with the buddy allocator alone and with the full system for SPECINT 2000 benchmarks, normalized by peak memory using the standard allocator without instrumentation.

as both the buddy allocator and pool allocation can be in principle applied independently on either system.

Next we measured the system using the SPECINT 2000 benchmarks. Figures 9 and 10 show the time and space overheads for SPECINT 2000 benchmarks.

We observe that the use of the buddy system has little effect on performance in average. The average runtime overhead of the full system with the benchmarks from SPECINT 2000 is 60%. `vpr` has the highest overhead of 127% because its frequent use of illegal pointers to fake base-one arrays invokes our slow path. We observed that adjusting the allocator to pad each allocation with 8 bytes from below, decreases the time overhead to 53% with only 5% added to the memory usage, although in general we are not interested in tuning the benchmarks like this. Interestingly, the overhead for `mcf` is a mere 16% compared to the 185% in [36] but the overhead of `gzip` is 55% compared to 15% in [36]. Such differences in performance are due to different levels of protection such as checking structure field indexing and checking dereferences, the effectiveness of different static analysis implementations in optimizing away checks, and the

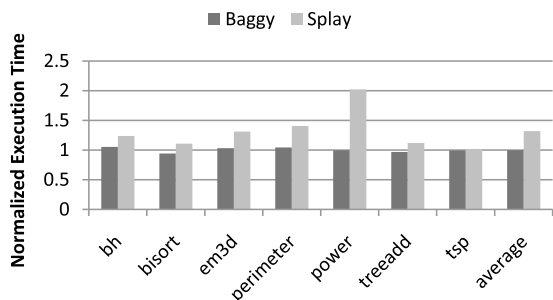


Figure 11: Execution time of baggy bounds checking versus using a splay tree for the Olden benchmark suite, normalized by the execution time using the standard system allocator without instrumentation. Benchmarks *mst* and *health* used too much memory and thrashed so their execution times are excluded.

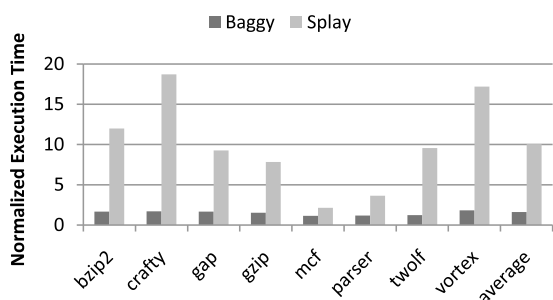


Figure 12: Execution time of baggy bounds checking versus using a splay tree for SPECINT 2000 benchmarks, normalized by the execution time using the standard system allocator without instrumentation.

different compilers used.

To isolate these effects, we also measured our system using the standard memory allocator and the splay tree implementation from previous systems [19, 30]. Figure 11 shows the time overhead for baggy bounds versus using a splay tree for the Olden benchmarks. The splay tree runs out of physical memory for the last two Olden benchmarks (*mst*, *health*) and slows down to a crawl, so we exclude them from the average of 30% for the splay tree. Figure 12 compares the time overhead against using a splay tree for the SPECINT 2000 benchmarks. The overhead of the splay tree exceeds 100% for all benchmarks, with an average of 900% compared to the average of 60% for baggy bounds checking.

Perhaps the most interesting result of our evaluation was space overhead. Previous solutions [19, 30, 15] do not report on the memory overheads of using splay trees, so we measured the memory overhead of our system using splay trees and compared it with the memory overhead of baggy bounds. Figure 13 shows that our system had

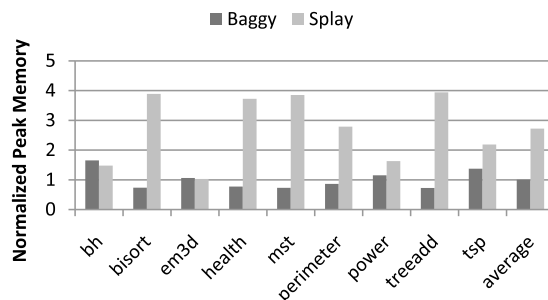


Figure 13: Peak memory use of baggy bounds checking versus using a splay tree for the Olden benchmark suite, normalized by peak memory using the standard allocator without instrumentation.

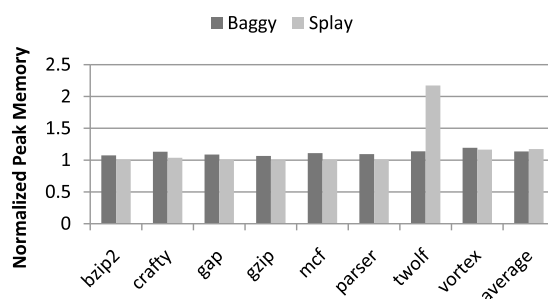


Figure 14: Peak memory use of baggy bounds checking versus using a splay tree for SPECINT 2000 benchmarks, normalized by peak memory using the standard allocator without instrumentation.

negligible memory overhead for Olden, as opposed to the splay tree version's 170% overhead. Clearly Olden's numerous small allocations stress the splay tree by forcing it to allocate an entry for each.

Indeed, we see in Figure 14 that its space overhead for most SPECINT 2000 benchmarks is very low. Nevertheless, the overhead of 15% for baggy bounds is less than the 20% average of the splay tree. Furthermore, the potential worst case of double the memory was not encountered for baggy bounds in any of our experiments, while the splay tree did exhibit greater than 100% overhead for one benchmark (*twolf*).

The memory overhead is also low, as expected, compared to approaches that track meta data for each pointer. Xu *et al.* [36] report 331% for Olden, and Nagarakatte *et al.* [23] report an average of 87% using a hash-table (and 64% using a contiguous array) over Olden and a subset of SPECINT and SPECFP, but more than about 260% (or about 170% using the array) for the pointer intensive Olden benchmarks alone. These systems suffer memory overheads per pointer in order to provide optional temporal protection [36] and sub-object protection [23] and

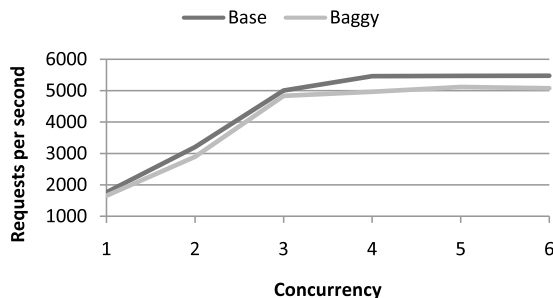


Figure 15: Throughput of Apache web server for varying numbers of concurrent requests.

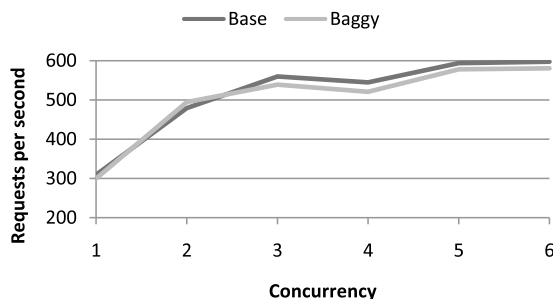


Figure 16: Throughput of NullHTTPD web server for varying numbers of concurrent requests.

it is interesting to contrast with them although they are not directly comparable.

4.2 Effectiveness

We evaluated the effectiveness of our system in preventing buffer overflows using the benchmark suite from [34]. The attacks required tuning to have any chance of success, because our system changes the stack frame layout and copies unsafe function arguments to local variables, but the benchmarks use the address of the first function argument to find the location of the return address they aim to overwrite.

Baggy bounds checking prevented 17 out of 18 buffer overflows in the suite. It failed, however, to prevent the overflow of an array inside a structure from overwriting a pointer inside the same structure. This limitation is also shared with other systems that detect memory errors at the level of memory blocks [19, 30, 36, 15].

4.3 Security Critical COTS Applications

Finally, to verify the usability of our approach, we built and measured a few additional larger and security critical

Program	KSLOC
openssl-0.9.8k	397
Apache-2.2.11	474
nullhttpd-0.5.1	2
libpng-1.2.5	36
SPECINT 2000	309
Olden	6
Total	1224

Table 1: Source lines of code in programs successfully built and run with baggy bounds.

COTS applications. Table 1 lists the total number of lines compiled in our experiments.

We built the OpenSSL toolkit version 0.9.8k [28] comprised of about 400 KSLOC, and executed its test suite measuring 10% time and 11% memory overhead.

Then we built and measured two web servers, Apache [31] and NullHTTPD [27]. Running NullHTTPD revealed three bounds violations similar to, and including, the one reported in [8]. We used the Apache benchmark utility with the keep-alive option to compare the throughput over a LAN connection of the instrumented and uninstrumented versions of both web servers. We managed to saturate the CPU by using the keep-alive option of the benchmarking utility to reuse connections for subsequent requests. We issued repeated requests for the servers' default pages and varied the number of concurrent clients until the throughput of the uninstrumented version leveled off (Figures 15 and 16). We verified that the server's CPU was saturated at this point, and measured a throughput decrease of 8% for Apache and 3% for NullHTTPD.

Finally, we built `libpng`, a notoriously vulnerability prone library that is widely used. We successfully ran its test program for 1000 PNG files between 1–2K found on a desktop machine, and measured an average runtime overhead of 4% and a peak memory overhead of 3.5%.

5 64-bit Architectures

In this section we verify and investigate ways to optimize our approach on 64 bit architectures. The key observation is that pointers in 64 bit architectures have spare bits to use. In Figure 17 (a) and (b) we see that current models of AMD64 processors use 48 out of 64 bits in pointers, and Windows further limit this to 43 bits for user space programs. Thus 21 bits in the pointer representation are not used. Next we describe two uses for these spare bits, and present a performance evaluation on AMD64.

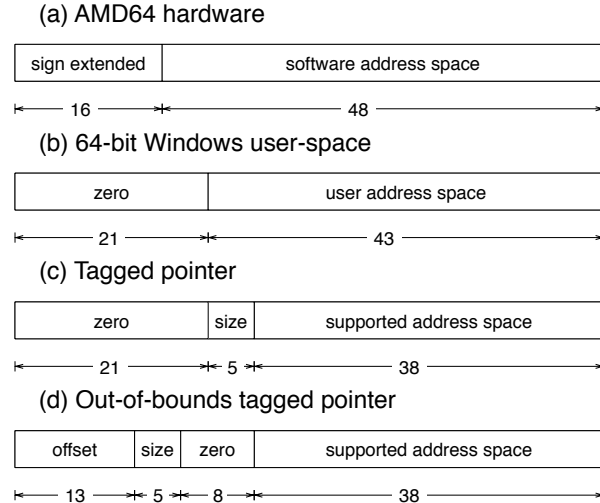


Figure 17: Use of pointer bits by AMD64 hardware, Windows applications, and baggy bounds tagged pointers.

5.1 Size Tagging

Since baggy bounds occupy less than a byte, they can fit in a 64 bit pointer's spare bits, removing the need for a separate data structure. These *tagged pointers* are similar to fat pointers in changing the pointer representation but have several advantages.

First, tagged pointers retain the size of regular pointers, avoiding fat pointers' register and memory waste. Moreover, their memory stores and loads are atomic, unlike fat pointers that break code relying on this. Finally, they preserve the memory layout of structures, overcoming the main drawback of fat pointers that breaks their interoperability with uninstrumented code.

For interoperability, we must also enable instrumented code to use pointers from uninstrumented code and vice versa. We achieve the former by interpreting the default zero value found in unused pointer bits as maximal bounds, so checks on pointers missing bounds succeed. The other direction is harder because we must avoid raising a hardware exception when uninstrumented code dereferences a tagged pointer.

We solved this using the paging hardware to map all addresses that differ only in their tag bits to the same memory. This way, unmodified binary libraries can use tagged pointers, and instrumented code avoids the cost of clearing the tag too.

As shown in Figure 17(c), we use 5 bits to encode the size, allowing objects up to 2^{32} bytes. In order to use the paging hardware, these 5 bits have to come from the 43 bits supported by the operating system, thus leaving 38

bits of address space for programs.

With 5 address bits used for the bounds, we need to map 32 different address regions to the same memory. We implemented this entirely in user space using the `CreateFileMapping` and `MapViewOfFileEx` Windows API functions to replace the process image, stack, and heap with a file backed by the system paging file and mapped at 32 different locations in the process address space.

We use the 5 bits effectively ignored by the hardware to store the size of memory allocations. For heap allocations, our `malloc`-style functions set the tags for pointers they return. For locals and globals, we instrument the address taking operator "&" to properly tag the resulting pointer. We store the bit complement of the size logarithm enabling interoperability with untagged pointers by interpreting their zero bit pattern as all bits set (representing a maximal allocation of 2^{32}).

```

extract
bounds
{
    mov rax, buf
    shr rax, 26h
    xor rax, 1fh
}

pointer
arithmetic
{
    char *p = buf[i];
}

bounds
check
{
    mov rbx, buf
    xor rbx, p
    shr rbx, al
    jz ok
    p = slowPath(buf, p)
    ok:
}

```

Figure 18: AMD64 code sequence inserted to check unsafe arithmetic with tagged pointers.

With the bounds encoded in pointers, there is no need for a memory lookup to check pointer arithmetic. Figure 18 shows the AMD64 code sequence for checking pointer arithmetic using a tagged pointer. First, we extract the encoded bounds from the source pointer by right shifting a copy to bring the tag to the bottom 8 bits of the register and xoring them with the value `0x1f` to recover the size logarithm by inverting the bottom 5 bits. Then we check that the result of the arithmetic is within bounds by xoring the source and result pointers, shifting the result by the tag stored in `al`, and checking for zero.

Similar to the table-based implementation of Section 3, out-of-bounds pointers trigger a bounds error to simplify the common case. To cause this, we zero the bits that were used to hold the size and save them using 5 more bits in the pointer, as shown in Figure 17(d).

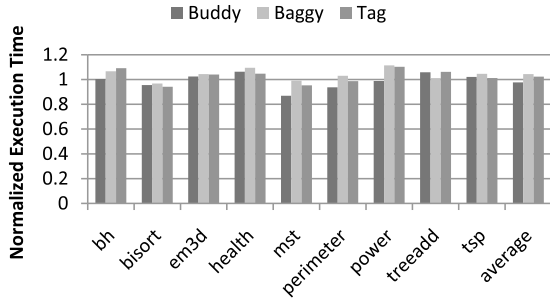


Figure 19: Normalized execution time on AMD64 with Olden benchmarks.

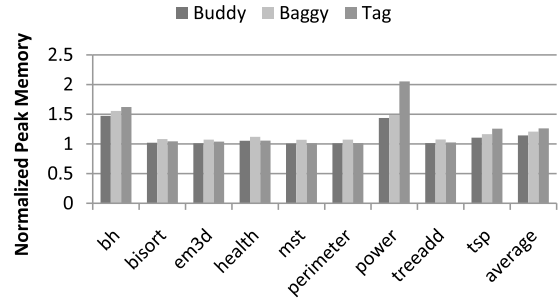


Figure 21: Normalized peak memory use on AMD64 with Olden benchmarks.

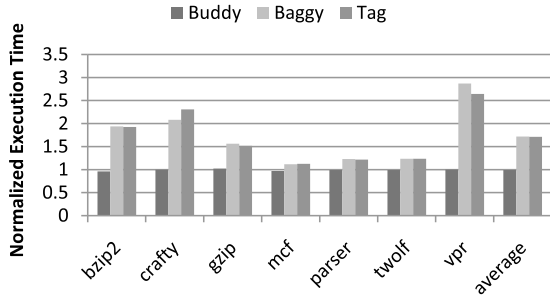


Figure 20: Normalized execution time on AMD64 with SPECINT 2000 benchmarks.

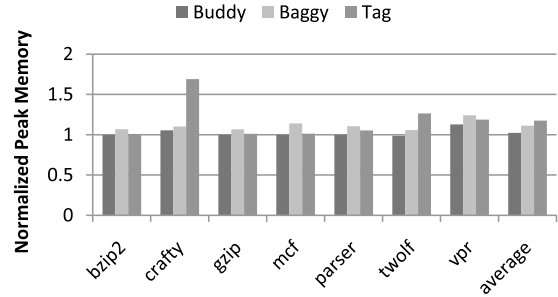


Figure 22: Normalized peak memory use on AMD64 with SPECINT 2000 benchmarks.

5.2 Out-Of-Bounds Offset

The spare bits can also store an offset that allows us to adjust an out-of-bounds pointer to recover the address of its referent object. We can use 13 bits for this offset, as shown in Figure 17(d). These bits can count slot or even allocation size multiples, increasing the supported out-of-bounds range to at least 2^{16} bytes above or below an allocation.

This technique does not depend on size tagging and can be used with a table instead. When looking up a pointer in the table, however, the top bits have to be masked off.

5.3 Evaluation

We evaluated baggy bounds checking on AMD64 using the subset of benchmarks from Section 4.1 that run unmodified on 64 bits. We measured the system using a contiguous array against the system using tagged pointers (Baggy and Tag in the figure legends respectively). We also measured the overhead using the buddy allocator only.

The multiple memory mappings complicated measuring memory use because Windows counts shared memory

multiple times in peak memory reports. To overcome this, we measured memory use without actually tagging the pointers, to avoid touching more than one address for the same memory, but with the memory mappings in place to account for at least the top level memory management overheads.

Figures 19 and 20 show the time overhead. The average using a table on 64-bits is 4% for Olden and 72% for SPECINT 2000—close to the 32-bit results of Section 3. Figures 21 and 22 show the space overhead. The average using a table is 21% for Olden and 11% for SPECINT 2000. Olden’s space overhead is higher than the 32-bit version; unlike the 32-bit case, the buddy allocator contributes to this overhead by 14% on average.

Tagged pointers are 1–2% faster on average than the table, and use about 5% less memory for most benchmarks, except a few ones such as *power* and *crafty*. These exceptions are because our prototype does not map pages to different addresses on demand, but instead maps 32 30-bit regions of virtual address space on program startup. Hence the fixed overhead is notable for these benchmarks because their absolute memory usage is low.

While we successfully implemented mapping multiple views entirely in user-space, a robust implementation would probably require kernel mode support. We feel

that the gains are too small to justify the complexity. However, using the spare bits to store an out-of-bounds offset is a good solution for tracking out-of-bounds pointers when using the referent object approach of Jones and Kelly [19].

6 Related Work

Many techniques have been proposed to detect memory errors in C programs. Static analysis techniques, e.g., [33, 21, 7], can detect defects before software ships and they do not introduce runtime overhead, but they can miss defects and raise false alarms.

Since static techniques do not remove all defects, they have been complemented with dynamic techniques. Debugging tools such as Purify [17] and Annelid [25] can find memory errors during testing. While these tools can be used without source code, they typically slow-down applications by a factor of 10 or more. Some dynamic techniques detect specific errors such as stack overflows [13, 16, 32] or format string exploits [12]; they have low overhead but they cannot detect all spatial memory errors. Techniques such as control-flow integrity [20, 1] or taint tracking (e.g. [10, 26, 11, 35]) detect broad classes of errors, but they do not provide general protection from spatial memory errors.

Some systems provide probabilistic protection from memory errors [5]. In particular, DieHard [4] increases heap allocation sizes by a random amount to make more out-of-bounds errors benign at a low performance cost. Our system also increases the allocation size but enforces the allocation bounds to prevent errors and also protects stack-allocated objects in addition to heap-allocated ones.

Several systems prevent all spatial memory errors in C programs. Systems such as SafeC [3], CCured [24], Cyclone [18], and the technique in Xu *et al.* [36] associate bounds information with each pointer. CCured [24] and Cyclone [18] are memory safe dialects of C. They extend the pointer representation with bounds information, i.e., they use a fat pointer representation, but this changes memory layout and breaks binary compatibility. Moreover, they require a significant effort to port applications to the safe dialects. For example, CCured required changing 1287 out of 6000 lines of code for the Olden benchmarks [15], and an average of 10% of the lines of code have to be changed when porting programs from C to Cyclone [34]. CCured has 28% average runtime overhead for the Olden benchmarks, which is significantly higher than the baggy bounds overhead. Xu *et al.* [36] track pointers to detect spatial errors as well

as temporal errors with additional overhead, thus their space overhead is proportional to the number of pointers. The average time overhead for spatial protection on the benchmarks we overlap is 73% versus 16% for baggy bounds with a space overhead of 273% versus 4%.

Other systems map any memory address within an allocated object to the bounds information for the object. Jones and Kelly [19] developed a backwards compatible bounds checking solution that uses a splay tree to map addresses to bounds. The splay tree is updated on allocation and deallocation, and operations on pointers are instrumented to lookup the bounds using an in-bounds pointer. The advantage over previous approaches using fat pointers is interoperability with code that was compiled without instrumentation. They increase the allocation size to support legal out-of-bounds pointers one byte beyond the object size. Baggy bounds checking offers similar interoperability with less time and space overhead, which we evaluated by using their implementation of splay trees with our system. CRED [30] improves on the solution of Jones and Kelly by adding support for tracking out-of-bounds pointers and making sure that they are never dereferenced unless they are brought within bounds again. Real programs often violate the C standard and contain such out-of-bounds pointers that may be saved to data structures. The performance overhead for programs that do not have out-of-bounds pointers is similar to Jones and Kelly if the same level of runtime checking is used, but the authors recommend only checking strings to lower the overhead to acceptable levels. For programs that do contain such out-of-bounds pointers the cost of tracking them includes scanning a hash-table on every dereference to remove entries for out-of-bounds pointers. Our solution is more efficient, and we propose ways to track common cases of out-of-bounds pointers that avoid using an additional data structure.

The fastest previous technique for bounds checking by Dhurjati *et al.* [15] is more than two times slower than our prototype. It uses inter-procedural data structure analysis to partition allocations into pools statically and uses a separate splay tree for each pool. They can avoid inserting some objects in the splay tree when the analysis finds that a pool is size-homogeneous. This should significantly reduce the memory usage of the splay tree compared to previous solutions, but unfortunately they do not report memory overheads. This work also optimizes the handling of out-of-bounds pointers in CRED [30] by relying on hardware memory protection to detect the dereference of out-of-bounds pointers.

The latest proposal, SoftBound [23], tracks bounds for each pointer to achieve sub-object protection. Sub-object

protection, however, may introduce compatibility problems with code using pointer arithmetic to traverse structures. SoftBound maintains interoperability by storing bounds in a hash table or a large contiguous array. Storing bounds for each pointer can lead to a worst case memory footprint as high as 300% for the hash-table version or 200% for the contiguous array. The average space overhead across Olden and a subset of SPECINT and SPECFP is 87% using a hash-table and 64% for the contiguous array, and the average runtime overhead for checking both reads and writes is 93% for the hash table and 67% for the contiguous array. Our average space overhead over Olden and SPECINT is 7.5% with an average time overhead of 32%.

Other approaches associate different kinds of metadata with memory regions to enforce safety properties. The technique in [37] detects some invalid pointers dereferences by marking all writable memory regions and preventing writes to non-writable memory; it reports an average runtime overhead of 97%. DFI [8] computes reaching definitions statically and enforces them at runtime. DFI has an average overhead of 104% on the SPEC benchmarks. WIT [2] computes the approximate set of objects written by each instruction and dynamically prevents writes to objects not in the set. WIT does not protect from invalid reads, and is subject to the precision of a points-to analysis when detecting some out-of-bounds errors. On the other hand, WIT can detect accesses to deallocated/unallocated objects and some accesses through dangling pointers to re-allocated objects in different analysis sets. WIT is six times faster than *baggy bounds checking* for SPECINT 2000, so it is also an attractive point in the error coverage/performance design space.

7 Limitations and Future Work

Our system shares some limitations with other solutions based on the referent object approach. Arithmetic on integers holding addresses is unchecked, casting an integer that holds an out-of-bounds address back to a pointer or passing an out-of-bounds pointer to unchecked code will break the program, and custom memory allocators reduce protection.

Our system does not address temporal memory safety violations (accesses through “dangling pointers” to re-allocated memory). Conservative garbage collection for C [6] is one way to address these but introduces its own compatibility issues and unpredictable overheads.

Our approach cannot protect from memory errors in sub-objects such as structure fields. To offer such protection,

a system must track the bounds of each pointer [23] and risk false alarms for some legal programs that use pointers to navigate across structure fields.

In Section 4 we found two programs using out-of-bounds pointers beyond the *slot_size*/2 bytes supported on 32-bits and one beyond the 2^{16} bytes supported on 64-bits. Unfortunately the real applications built in Section 4.3 were limited to software we could readily port to the Windows toolchain; wide use will likely encounter occasional problems with out-of-bounds pointers, especially on 32-bit systems. We plan to extend our system to support all out-of-bounds pointers using the data structure from [30], but take advantage of the more efficient mechanisms we described for the common cases. To solve the delayed deallocation problem discussed in Section 6 and deallocate entries as soon as the out-of-bounds pointer is deallocated, we can track out-of-bounds pointers using the pointer’s address instead of the pointer’s referent object’s address. (Similar to the approach [23] takes for all pointers.) To optimize scanning this data structure on every deallocation we can use an array with an entry for every few memory pages. A single memory read from this array on deallocation (e.g. on function exit) is sufficient to confirm the data structure has no entries for a memory range. This is the common case since most out-of-bounds pointers are handled by the other mechanisms we described in this paper.

Our prototype uses a simple intra-procedural analysis to find safe operations and does not eliminate redundant checks. We expect that integrating state of the art analyses to reduce the number of checks will further improve performance.

Finally, our approach tolerates harmless bound violations making it less suitable for debugging than slower techniques that can uncover these errors. On the other hand, being faster makes it more suitable for production runs, and tolerating faults in production runs may be desired [29].

8 Conclusions

Attacks that exploit out-of-bounds errors in C and C++ continue to be a serious security problem. We presented *baggy bounds checking*, a backwards-compatible bounds checking technique that implements efficient bounds checks. It improves the performance of bounds checks by checking allocation bounds instead of object bounds and by using a binary buddy allocator to constrain the size and alignment of allocations to powers of 2. These constraints enable a concise representation for allocation bounds and let *baggy bounds checking* store this infor-

mation in an array that can be looked up and maintained efficiently. Our experiments show that replacing a splay tree, which was used to store bounds information in previous systems, by our array reduces time overhead by an order of magnitude without increasing space overhead.

We believe *baggy bounds checking* can be used in practice to harden security-critical applications because it has low overhead, it works on unmodified C and C++ programs, and it preserves binary compatibility with uninstrumented libraries. For example, we were able to compile the Apache Web server with *baggy bounds checking* and the throughput of the hardened version of the server decreases by only 8% relative to an uninstrumented version.

Acknowledgments

We thank our shepherd R. Sekar, the anonymous reviewers, and the members of the Networks and Operating Systems group at Cambridge University for comments that helped improve this paper. We also thank Dinakar Dhurjati and Vikram Adve for their communication.

References

- [1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, 2005.
- [2] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing memory error exploits with WIT. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, 2008.
- [3] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1994.
- [4] Emery D. Berger and Benjamin G. Zorn. DieHard: probabilistic memory safety for unsafe languages. In *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, 2006.
- [5] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th USENIX Security Symposium*, 2005.
- [6] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. In *Software Practice & Experience*, 1988.
- [7] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. In *Software Practice & Experience*, 2000.
- [8] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.
- [9] Jim Chow, Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. Shredding your garbage: reducing data lifetime through secure deallocation. In *Proceedings of the 14th USENIX Security Symposium*, 2005.
- [10] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Can we contain Internet worms? In *Proceedings of the Third Workshop on Hot Topics in Networks (HotNets-III)*, 2004.
- [11] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: end-to-end containment of internet worms. In *Proceedings of the 20th ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2005.
- [12] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Mike Frantzen, and Jamie Lokier. FormatGuard: automatic protection from printf format string vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, 2001.
- [13] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, 1998.
- [14] John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram Adve. Secure virtual architecture: a safe execution environment for commodity operating systems. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2007.
- [15] Dinakar Dhurjati and Vikram Adve. Backwards-compatible array bounds checking for C with very low overhead. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, 2006.

- [16] Hiroaki Etoh and Kunikazu Yoda. Protecting from stack-smashing attacks. <http://www.trl.ibm.com/projects/security/ssp/main.html>.
- [17] Reed Hasting and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, 1992.
- [18] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *Proceedings of the General Track of the USENIX Annual Conference*, 2002.
- [19] Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the 3rd International Workshop on Automatic Debugging (AADEBUDG)*, 1997.
- [20] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, 2002.
- [21] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, 2001.
- [22] Microsoft. Phoenix compiler framework. <http://connect.microsoft.com/Phoenix>.
- [23] Santosh Nagarakatte, Jianzhou Zhao, Milo Martin, and Steve Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for C. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.
- [24] George C. Necula, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2002.
- [25] Nicholas Nethercote and Jeremy Fitzhardinge. Bounds-checking entire programs without recompiling. In *Informal Proceedings of the Second Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management (SPACE)*, 2004.
- [26] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis and signature generation of exploits on commodity software. In *Proceedings of the 12th Network and Distributed System Security Symposium (NDSS)*, 2005.
- [27] NullLogic. Null HTTPd. <http://nullwebmail.sourceforge.net/httpd>.
- [28] OpenSSL Toolkit. <http://www.openssl.org>.
- [29] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebe, Jr. Enhancing server availability and security through failure-oblivious computing. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.
- [30] Olatunji Ruwase and Monica S. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the 11th Network and Distributed System Security Symposium (NDSS)*, 2004.
- [31] The Apache Software Foundation. The Apache HTTP Server Project. <http://httpd.apache.org>.
- [32] Vendicator. StackShield. <http://www.angelfire.com/sk/stackshield>.
- [33] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of the 7th Network and Distributed System Security Symposium (NDSS)*, 2000.
- [34] John Wilander and Mariam Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the 10th Network and Distributed System Security Symposium (NDSS)*, 2003.
- [35] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks. In *Proceedings of the 15th USENIX Security Symposium*, 2006.
- [36] Wei Xu, Daniel C. DuVarney, and R. Sekar. An efficient and backwards-compatible transformation to ensure memory safety of C programs. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT/FSE)*, 2004.
- [37] Suan Hsi Yong and Susan Horwitz. Protecting C programs from attacks via invalid pointer dereferences. In *Proceedings of the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2003.

Dynamic Test Generation To Find Integer Bugs in x86 Binary Linux Programs

David Molnar
UC Berkeley

Xue Cong Li
UC Berkeley

David A. Wagner
UC Berkeley

Abstract

Recently, integer bugs, including integer overflow, width conversion, and signed/unsigned conversion errors, have risen to become a common root cause for serious security vulnerabilities. We introduce new methods for discovering integer bugs using dynamic test generation on x86 binaries, and we describe key design choices in efficient symbolic execution of such programs. We implemented our methods in a prototype tool *SmartFuzz*, which we use to analyze Linux x86 binary executables. We also created a reporting service, *metafuzz.com*, to aid in triaging and reporting bugs found by *SmartFuzz* and the black-box fuzz testing tool *zzuf*. We report on experiments applying these tools to a range of software applications, including the *mplayer* media player, the *exiv2* image metadata library, and *ImageMagick convert*. We also report on our experience using *SmartFuzz*, *zzuf*, and *metafuzz.com* to perform testing at scale with the Amazon Elastic Compute Cloud (EC2). To date, the *metafuzz.com* site has recorded more than 2,614 test runs, comprising 2,361,595 test cases. Our experiments found approximately 77 total distinct bugs in 864 compute hours, costing us an average of \$2.24 per bug at current EC2 rates. We quantify the overlap in bugs found by the two tools, and we show that *SmartFuzz* finds bugs missed by *zzuf*, including one program where *SmartFuzz* finds bugs but *zzuf* does not.

1 Introduction

Integer overflow bugs recently became the second most common bug type in security advisories from OS vendors [10]. Unfortunately, traditional static and dynamic analysis techniques are poorly suited to detecting integer-related bugs. In this paper, we argue that *dynamic test generation* is better suited to finding such bugs, and we develop new methods for finding a broad class of integer bugs with this approach. We have implemented these methods in a new tool, *SmartFuzz*, that analyzes traces from commodity Linux x86 programs.

Integer bugs result from a mismatch between machine arithmetic and mathematical arithmetic. For example, machine arithmetic has bounded precision; if an expression has a value greater than the maximum integer that can be represented, the value wraps around to fit in machine precision. This can cause the value stored to be smaller than expected by the programmer. If, for example, a wrapped value is used as an argument to *malloc*, the result is an object that is smaller than expected, which can lead to a buffer overflow later if the programmer is not careful. This kind of bug is often known as an integer overflow bug. In Section 2 we describe two other classes of integer bugs: *width conversions*, in which converting from one type of machine integer to another causes unexpected changes in value, and *signed/unsigned conversions*, in which a value is treated as both a signed and an unsigned integer. These kinds of bugs are pervasive and can, in many cases, cause serious security vulnerabilities. Therefore, eliminating such bugs is important for improving software security.

While new code can partially or totally avoid integer bugs if it is constructed appropriately [19], it is also important to find and fix bugs in legacy code. Previous approaches to finding integer bugs in legacy code have focused on static analysis or runtime checks. Unfortunately, existing static analysis algorithms for finding integer bugs tend to generate many false positives, because it is difficult to statically reason about integer values with sufficient precision. Alternatively, one can insert runtime checks into the application to check for overflow or non-value-preserving width conversions, and raise an exception if they occur. One problem with this approach is that many overflows are benign and harmless. Throwing an exception in such cases prevents the application from functioning and thus causes false positives. Furthermore, occasionally the code intentionally relies upon overflow semantics; e.g., cryptographic code or fast hash functions. Such code is often falsely flagged by static analysis or runtime checks. In summary, both static analysis and

runtime checking tend to suffer from either many false positives or many missed bugs.

In contrast, *dynamic test generation* is a promising approach for avoiding these shortcomings. Dynamic test generation, a technique introduced by Godefroid et al. and Engler et al. [13, 7], uses *symbolic execution* to generate new test cases that expose specifically targeted behaviors of the program. Symbolic execution works by collecting a set of constraints, called the *path condition*, that model the values computed by the program along a single path through the code. To determine whether there is any input that could cause the program to follow that path of execution and also violate a particular assertion, we can add to the path condition a constraint representing that the assertion is violated and feed the resulting set of constraints to a solver. If the solver finds any solution to the resulting constraints, we can synthesize a new test case that will trigger an assertion violation. In this way, symbolic execution can be used to discover test cases that cause the program to behave in a specific way.

Our main approach is to use symbolic execution to construct test cases that trigger arithmetic overflows, non-value-preserving width conversions, or dangerous signed/unsigned conversions. Then, we run the program on these test cases and use standard tools that check for buggy behavior to recognize bugs. We only report test cases that are verified to trigger incorrect behavior by the program. As a result, we have confidence that all test cases we report are real bugs and not false positives.

Others have previously reported on using dynamic test generation to find some kinds of security bugs [8, 15]. The contribution of this paper is to show how to extend those techniques to find integer-related bugs. We show that this approach is effective at finding many bugs, without the false positives endemic to prior work on static analysis and runtime checking.

The ability to eliminate false positives is important, because false positives are time-consuming to deal with. In slogan form: false positives in static analysis waste the programmer's time; false positives in runtime checking waste the end user's time; while false positives in dynamic test generation waste the tool's time. Because an hour of CPU time is much cheaper than an hour of a human's time, dynamic test generation is an attractive way to find and fix integer bugs.

We have implemented our approach to finding integer bugs in *SmartFuzz*, a tool for performing symbolic execution and dynamic test generation on Linux x86 applications. SmartFuzz works with binary executables directly, and does not require or use access to source code. Working with binaries has several advantages, most notably that we can generate tests directly from shipping binaries. In particular, we do not need to modify the build process for a program under test, which has been

a pain point for static analysis tools [9]. Also, this allows us to perform whole-program analysis: we can find bugs that arise due to interactions between the application and libraries it uses, even if we don't have source code for those libraries. Of course, working with binary traces introduces special challenges, most notably the sheer size of the traces and the lack of type information that would be present in the source code. We discuss the challenges and design choices in Section 4.

In Section 5 we describe the techniques we use to generate test cases for integer bugs in dynamic test generation. We discovered that these techniques find many bugs, too many to track manually. To help us prioritize and manage these bug reports and streamline the process of reporting them to developers, we built *Metafuzz*, a web service for tracking test cases and bugs (Section 6). Metafuzz helps minimize the amount of human time required to find high-quality bugs and report them to developers, which is important because human time is the most expensive resource in a testing framework. Finally, Section 7 presents an empirical evaluation of our techniques and discusses our experience with these tools.

The contributions of this paper are the following:

- We design novel algorithms for finding signed/unsigned conversion vulnerabilities using symbolic execution. In particular, we develop a novel type inference approach that allows us to detect which values in an x86 binary trace are used as signed integers, unsigned integers, or both. We discuss challenges in scaling such an analysis to commodity Linux media playing software and our approach to these challenges.
- We extend the range of integer bugs that can be found with symbolic execution, including integer overflows, integer underflows, width conversions, and signed/unsigned conversions. No prior symbolic execution tool has included the ability to detect all of these kinds of integer vulnerabilities.
- We implement these methods in *SmartFuzz*, a tool for symbolic execution and dynamic test generation of x86 binaries on Linux. We describe key challenges in symbolic execution of commodity Linux software, and we explain design choices in SmartFuzz motivated by these challenges.
- We report on the bug finding performance of SmartFuzz and compare SmartFuzz to the zzuf black box fuzz testing tool. The zzuf tool is a simple, yet effective, *fuzz testing* program which randomly mutates a given seed file to find new test inputs, without any knowledge or feedback from the target program. We have tested a broad range of commodity Linux software, including the media players

ffmpeg, the ImageMagick convert tool, and the exiv2 TIFF metadata parsing library. This software comprises over one million lines of source code, and our test cases result in symbolic execution of traces that are millions of x86 instructions in length.

- We identify challenges with reporting bugs at scale, and introduce several techniques for addressing these challenges. For example, we present evidence that a simple stack hash is not sufficient for grouping test cases to avoid duplicate bug reports, and then we develop a *fuzzy stack hash* to solve these problems. Our experiments find approximately 77 total distinct bugs in 864 compute hours, giving us an average cost of \$2.24 per bug at current Amazon EC2 rates. We quantify the overlap in bugs found by the two tools, and we show that SmartFuzz finds bugs missed by zzuf, including one program where SmartFuzz finds bugs but zzuf does not.

Between June 2008 and November 2008, Metafuzz has processed over 2,614 test runs from both SmartFuzz and the zzuf black box fuzz testing tool [16], comprising 2,361,595 test cases. To our knowledge, this is the largest number of test runs and test cases yet reported for dynamic test generation techniques. We have released our code under the GPL version 2 and BSD licenses¹. Our vision is a service that makes it easy and inexpensive for software projects to find integer bugs and other serious security relevant code defects using dynamic test generation techniques. Our work shows that such a service is possible for a large class of commodity Linux programs.

2 Integer Bugs

We now describe the three main classes of integer bugs we want to find: integer overflow/underflow, width conversions, and signed/unsigned conversion errors [2]. All three classes of bugs occur due to the mismatch between machine arithmetic and arithmetic over unbounded integers.

Overflow/Underflow. Integer overflow (and underflow) bugs occur when an arithmetic expression results in a value that is larger (or smaller) than can be represented by the machine type. The usual behavior in this case is to silently “wrap around,” e.g. for a 32-bit type, reduce the value modulo 2^{32} . Consider the function `badalloc` in Figure 1. If the multiplication `sz * n` overflows, the allocated buffer may be smaller than expected, which can lead to a buffer overflow later.

Width Conversions. Converting a value of one integral type to a wider (or narrower) integral type which has a

```
char *badalloc(int sz, int n) {
    return (char *) malloc(sz * n);
}

void badcpy(Int16 n, char *p, char *q) {
    UInt32 m = n;
    memcpy(p, q, m);
}

void badcpy2(int n, char *p, char *q) {
    if (n > 800)
        return;
    memcpy(p, q, n);
}
```

Figure 1: Examples of three types of integer bugs.

different range of values can introduce *width conversion* bugs. For instance, consider `badcpy` in Figure 1. If the first parameter is negative, the conversion from `Int16` to `UInt32` will trigger sign-extension, causing `m` to be very large and likely leading to a buffer overflow. Because `memcpy`’s third argument is declared to have type `size_t` (which is an unsigned integer type), even if we passed `n` directly to `memcpy` the implicit conversion would still make this buggy. Width conversion bugs can also arise when converting a wider type to a narrower type.

Signed/Unsigned Conversion. Lastly, converting a signed integer type to an unsigned integer type of the same width (or vice versa) can introduce bugs, because this conversion can change a negative number to a large positive number (or vice versa). For example, consider `badcpy2` in Figure 1. If the first parameter `n` is a negative integer, it will pass the bounds check, then be promoted to a large unsigned integer when passed to `memcpy`. `memcpy` will copy a large number of bytes, likely leading to a buffer overflow.

3 Related Work

An earlier version of SmartFuzz and the Metafuzz web site infrastructure described in this paper were used for previous work that compares dynamic test generation with black-box fuzz testing by different authors [1]. That previous work does not describe the SmartFuzz tool, its design choices, or the Metafuzz infrastructure in detail. Furthermore, this paper works from new data on the effectiveness of SmartFuzz, except for an anecdote in our “preliminary experiences” section. We are not aware of other work that directly compares dynamic test generation with black-box fuzz testing on a scale similar to ours.

The most closely related work on integer bugs is Godefroid et al. [15], who describe dynamic test generation with bug-seeking queries for integer overflow, underflow, and some narrowing conversion errors in the context of the SAGE tool. Our work looks at a wider

¹<http://www.sf.net/projects/catchconv>

range of narrowing conversion errors, and we consider signed/unsigned conversion while their work does not. The EXE and KLEE tools also use integer overflow to prioritize different test cases in dynamic test generation, but they do not break out results on the number of bugs found due to this heuristic [8, 6]. The KLEE system also focuses on scaling dynamic test generation, but in a different way. While we focus on a few “large” programs in our results, KLEE focuses on high code coverage for over 450 smaller programs, as measured by trace size and source lines of code. These previous works also do not address the problem of type inference for integer types in binary traces.

IntScope is a static binary analysis tool for finding integer overflow bugs [28]. IntScope translates binaries to an intermediate representation, then it checks lazily for potentially harmful integer overflows by using symbolic execution for data that flows into “taint sinks” defined by the tool, such as memory allocation functions. SmartFuzz, in contrast, *eagerly* attempts to generate new test cases that cause an integer bug at the point in the program where such behavior could occur. This difference is due in part to the fact that IntScope reports errors to a programmer directly, while SmartFuzz filters test cases using a tool such as memcheck. As we argued in the Introduction, such a filter allows us to employ aggressive heuristics that may generate many test cases. Furthermore, while IntScope renders signed and unsigned comparisons in their intermediate representation by using hints from the x86 instruction set, they do not explicitly discuss how to use this information to perform type inference for signed and unsigned types, nor do they address the issue of scaling such inference to traces with millions of instructions. Finally, IntScope focuses only on integer overflow errors, while SmartFuzz covers underflow, narrowing conversion, and signed/unsigned conversion bugs in addition.

The dynamic test generation approach we use was introduced by Godefroid et al. [13] and independently by Cadar and Engler [7]. The SAGE system by Godefroid et al. works, as we do, on x86 binary programs and uses a generational search, but SAGE makes several different design choices we explain in Section 4. Lanzi et al. propose a design for dynamic test generation of x86 binaries that uses static analysis of loops to assist the solver, but their implementation is preliminary [17]. KLEE, in contrast, works with the intermediate representation generated by the Low-Level Virtual Machine target for gcc [6]. Larson and Austin applied symbolic range analysis to traces of programs to look for potential buffer overflow attacks, although they did not attempt to synthesize crashing inputs [18]. The BitBlaze [5] infrastructure of Song et al. also performs symbolic execution of x86 binaries, but their focus is on malware and signa-

ture generation, not on test generation.

Other approaches to integer bugs include static analysis and runtime detection. The Microsoft Prefast tool uses static analysis to warn about intraprocedural integer overflows [21]. Both Microsoft Visual C++ and gcc can add runtime checks to catch integer overflows in arguments to malloc and terminate a program. Brumley et al. provide rules for such runtime checks and show they can be implemented with low overhead on the x86 architecture by using jumps conditioned on the overflow bit in EFLAGS [4]. Both of these approaches fail to catch signed/unsigned conversion errors. Furthermore, both static analysis and runtime checking for overflow will flag code that is correct but relies on overflow semantics, while our approach only reports test cases in case of a crash or a Valgrind error report.

Blexim gives an introduction to integer bugs [3]. Fuzz testing has received a great deal of attention since its original introduction by Miller et al [22]. Notable public demonstrations of fuzzing’s ability to find bugs include the Month of Browser Bugs and Month of Kernel Bugs [23, 20]. DeMott surveys recent work on fuzz testing, including the autodafe fuzzer, which uses libgdb to instrument functions of interest and adjust fuzz testing based on those functions’ arguments [11, 27].

Our Metafuzz infrastructure also addresses issues not treated in previous work on test generation. First, we make *bug bucketing* a first-class problem and we introduce a *fuzzy stack hash* in response to developer feedback on bugs reported by Metafuzz. The SAGE paper reports bugs by stack hash, and KLEE reports on using the line of code as a bug bucketing heuristic, but we are not aware of other work that uses a fuzzy stack hash. Second, we report techniques for reducing the amount of human time required to process test cases generated by fuzzing and improve the quality of our error reports to developers; we are not aware of previous work on this topic. Such techniques are vitally important because human time is the most expensive part of a test infrastructure. Finally, Metafuzz uses on-demand computing with the Amazon Elastic Compute Cloud, and we explicitly quantify the cost of each bug found, which was not done in previous work.

4 Dynamic Test Generation

We describe the architecture of *SmartFuzz*, a tool for dynamic test generation of x86 binary programs on Linux. Dynamic test generation on x86 binaries—without access to source code—raises special challenges. We discuss these challenges and motivate our fundamental design choices.

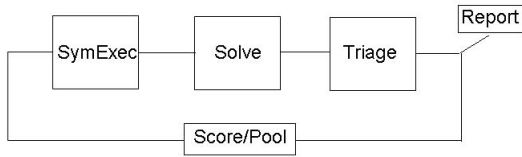


Figure 2: Dynamic test generation includes four stages: symbolic execution, solving to obtain new test cases, then triage to determine whether to report a bug or score the test case for addition to the pool of unexplored test cases.

4.1 Architecture

The SmartFuzz architecture is as follows: First, we add one or more test cases to a pool. Each test case in the pool receives a score given by the number of new basic blocks seen when running the target program on the test case. By “new” we mean that the basic block has not been observed while scoring any previous test case; we identify basic blocks by the instruction pointer of their entry point.

In each iteration of test generation, we choose a high-scoring test case, execute the program on that input, and use symbolic execution to generate a set of constraints that record how each intermediate value computed by the program relates to the inputs in the test case. SmartFuzz implements the symbolic execution and scoring components using the Valgrind binary analysis framework, and we use STP [12] to solve constraints.

For each symbolic branch, SmartFuzz adds a constraint that tries to force the program down a different path. We then query the constraint solver to see whether there exists any solution to the resulting set of constraints; if there is, the solution describes a new test case. We refer to these as *coverage queries* to the constraint solver.

SmartFuzz also injects constraints that are satisfied if a condition causing an error or potential error is satisfied (e.g., to force an arithmetic calculation to overflow). We then query the constraint solver; a solution describes a test case likely to cause an error. We refer to these as *bug-seeking queries* to the constraint solver. Bug-seeking queries come in different *types*, depending on the specific error they seek to exhibit in the program.

Both coverage and bug-seeking queries are explored in a *generational search* similar to the SAGE tool [14]. Each query from a symbolic trace is solved in turn, and new test cases created from successfully solved queries. A single symbolic execution therefore leads to many coverage and bug-seeking queries to the constraint solver, which may result in many new test cases.

We *triage* each new test case as it is generated, i.e. we

determine if it exhibits a bug. If so, we report the bug; otherwise, we add the test case to the pool for scoring and possible symbolic execution. For triage, we use Valgrind memcheck on the target program with each test case, which is a tool that observes concrete execution looking for common programming errors [26]. We record any test case that causes the program to crash or triggers a memcheck warning.

We chose memcheck because it checks a variety of properties, including reads and writes to invalid memory locations, memory leaks, and use of uninitialized values. Re-implementing these analyses as part of the SmartFuzz symbolic execution tool would be wasteful and error-prone, as the memcheck tool has had the benefit of multiple years of use in large-scale projects such as Firefox and OpenOffice. The memcheck tool is known as a tool with a low false positive rate, as well, making it more likely that developers will pay attention to bugs reported by memcheck. Given a memcheck error report, developers do not even need to know that associated test case was created by SmartFuzz.

We do not attempt to classify the bugs we find as exploitable or not exploitable, because doing so by hand for the volume of test cases we generate is impractical. Many of the bugs found by memcheck are memory safety errors, which often lead to security vulnerabilities. Writes to invalid memory locations, in particular, are a red flag. Finally, to report bugs we use the Metafuzz framework described in Section 6.

4.2 Design Choices

Intermediate Representation. The sheer size and complexity of the x86 instruction set poses a challenge for analyzing x86 binaries. We decided to translate the underlying x86 code on-the-fly to an intermediate representation, then map the intermediate representation to symbolic formulas. Specifically, we used the Valgrind binary instrumentation tool to translate x86 instructions into VEX, the Valgrind intermediate representation [24]. The BitBlaze system works similarly, but with a different intermediate representation [5]. Details are available in an extended version of this paper².

Using an intermediate representation offers several advantages. First, it allows for a degree of platform independence: though we support only x86 in our current tool, the VEX library also supports the AMD64 and PowerPC instruction sets, with ARM support under active development. Adding support for these additional architectures requires only adding support for a small number of additional VEX instructions, not an entirely new instruction set from scratch. Second, the VEX library generates IR that satisfies the single static assignment property and

²<http://www.cs.berkeley.edu/~dmolnar/usenix09-full1.pdf>

performs other optimizations, which makes the translation from IR to formulas more straightforward. Third, and most importantly, this choice allowed us to outsource the pain of dealing with the minutiae of the x86 instruction set to the VEX library, which has had years of production use as part of the Valgrind memory checking tool. For instance, we don't need to explicitly model the EFLAGS register, as the VEX library translates it to boolean operations. The main shortcoming with the VEX IR is that a single x86 instruction may expand to five or more IR instructions, which results in long traces and correspondingly longer symbolic formulas.

Online Constraint Generation. SmartFuzz uses *online* constraint generation, in which constraints are generated while the program is running. In contrast, SAGE (another tool for dynamic test generation) uses *offline* constraint generation, where the program is first traced and then the trace is replayed to generate constraints [14]. Offline constraint generation has several advantages: it is not sensitive to concurrency or nondeterminism in system calls; tracing has lower runtime overhead than constraint generation, so can be applied to running systems in a realistic environment; and, this separation of concerns makes the system easier to develop and debug, not least because trace replay and constraint generation is reproducible and deterministic. In short, offline constraint generation has important software engineering advantages.

SmartFuzz uses online constraint generation primarily because, when the SmartFuzz project began, we were not aware of an available offline trace-and-replay framework with an intermediate representation comparable to VEX. Today, O'Callahan's *chronicle-recorder* could provide a starting point for a VEX-based offline constraint generation tool [25].

Memory Model. Other symbolic execution tools such as EXE and KLEE model memory as a set of symbolic arrays, with one array for each allocated memory object. We do not. Instead, for each load or store instruction, we first concretize the memory address before accessing the symbolic heap. In particular, we keep a map M from concrete memory addresses to symbolic values. If the program reads from concrete address a , we retrieve a symbolic value from $M(a)$. Even if we have recorded a symbolic expression a associated with this address, the symbolic address is ignored. Note that the value of a is known at constraint generation time and hence becomes (as far as the solver is concerned) a constant. Store instructions are handled similarly.

While this approach sacrifices precision, it scales better to large traces. We note that the SAGE tool adopts a similar memory model. In particular, concretizing addresses generates symbolic formulas that the constraint solver can solve much more efficiently, because the

solver does not need to reason about aliasing of pointers.

Only Tainted Data is Symbolic. We track the taint status of every byte in memory. As an optimization, we do not store symbolic information for untainted memory locations, because by definition untainted data is not dependent upon the untrusted inputs that we are trying to vary. We have found that only a tiny fraction of the data processed along a single execution path is tainted. Consequently, this optimization greatly reduces the size of our constraint systems and reduces the memory overhead of symbolic execution.

Focus on Fuzzing Files. We decided to focus on single-threaded programs, such as media players, that read a file containing untrusted data. Thus, a test case is simply the contents of this file, and SmartFuzz can focus on generating candidate files. This simplifies the symbolic execution and test case generation infrastructure, because there are a limited number of system calls that read from this file, and we do not need to account for concurrent interactions between threads in the same program. We know of no fundamental barriers, however, to extending our approach to multi-threaded and network-facing programs.

Our implementation associates a symbolic input variable with each byte of the input file. As a result, SmartFuzz cannot generate test cases with more bytes than are present in the initial seed file.

Multiple Cooperating Analyses. Our tool is implemented as a series of independent cooperating analyses in the Valgrind instrumentation framework. Each analysis adds its own instrumentation to a basic block during translation and exports an interface to the other analyses. For example, the instrumentation for tracking taint flow, which determines the IR instructions to treat as symbolic, exports an interface that allows querying whether a specific memory location or temporary variable is symbolic. A second analysis then uses this interface to determine whether or not to output STP constraints for a given IR instruction.

The main advantage of this approach is that it makes it easy to add new features by adding a new analysis, then modifying our core constraint generation instrumentation. Also, this decomposition enabled us to extract our taint-tracking code and use it in a different project with minimal modifications, and we were able to implement the binary type inference analysis described in Section 5, replacing a different earlier version, without changing our other analyses.

Optimize in Postprocessing. Another design choice was to output constraints that are as "close" as possible to the intermediate representation, performing only limited optimizations on the fly. For example, we implement the "related constraint elimination," as introduced by tools

such as EXE and SAGE [8, 14], as a post-processing step on constraints created by our tool. We then leave it up to the solver to perform common subexpression elimination, constant propagation, and other optimizations. The main benefit of this choice is that it simplifies our constraint generation. One drawback of this choice is that current solvers, including STP, are not yet capable of “remembering” optimizations from one query to the next, leading to redundant work on the part of the solver. The main drawback of this choice, however, is that while after optimization each individual query is small, the total symbolic trace containing all queries for a program can be several gigabytes. When running our tool on a 32-bit host machine, this can cause problems with maximum file size for a single file or maximum memory size in a single process.

5 Techniques for Finding Integer Bugs

We now describe the techniques we use for finding integer bugs.

Overflow/Underflow. For each arithmetic expression that could potentially overflow or underflow, we emit a constraint that is satisfied if the overflow or underflow occurs. If our solver can satisfy these constraints, the resulting input values will likely cause an underflow or overflow, potentially leading to unexpected behavior.

Width Conversions. For each conversion between integer types, we check whether it is possible for the source value to be outside the range of the target value by adding a constraint that’s satisfied when this is the case and then applying the constraint solver. For conversions that may sign-extend, we use the constraint solver to search for a test case where the high bit of the source value is non-zero.

Signed/Unsigned Conversions. Our basic approach is to try to reconstruct, from the x86 instructions executed, signed/unsigned type information about all integral values. This information is present in the source code but not in the binary, so we describe an algorithm to infer this information automatically.

Consider four types for integer values: “Top,” “Signed,” “Unsigned,” or “Bottom.” Here, “Top” means the value has not been observed in the context of a signed or unsigned integer; “Signed” means that the value has been used as a signed integer; “Unsigned” means the value has been used as an unsigned integer; and “Bottom” means that the value has been used inconsistently as both a signed and unsigned integer. These types form a four-point lattice. Our goal is to find symbolic program values that have type “Bottom.” These values are candidates for signed/unsigned conversion errors. We then attempt to synthesize an input that forces these values to be negative.

We associate every instance of every temporary vari-

```
int main(int argc, char** argv) {
    char * p = malloc(800);
    char * q = malloc(800);
    int n;
    n = atoi(argv[1]);
    if (n > 800)
        return;
    memcpy(p, q, n);
    return 0;
}
```

Figure 3: A simple test case for dynamic type inference and query generation. The signed comparison $n > 800$ and unsigned `size_t` argument to `memcpy` assign the type “Bottom” to the value associated with `n`. When we solve for an input that makes `n` negative, we obtain a test case that reveals the error.

able in the Valgrind intermediate representation with a type. Every variable in the program starts with type Top. During execution we add *type constraints* to the type of each value. For x86 binaries, the sources of type constraints are signed and unsigned comparison operators: e.g., a signed comparison between two values causes both values to receive the “Signed” type constraint. We also add unsigned type constraints to values used as the length argument of `memcpy` function, which we can detect because we know the calling convention for x86 and we have debugging symbols for glibc. While the x86 instruction set has additional operations, such as `IMUL` that reveal type information about their operands, we do not consider these; this means only that we may incorrectly under-constrain the types of some values.

Any value that has received both a signed and unsigned type constraint receives the type Bottom. After adding a type constraint, we check to see if the type of a value has moved to Bottom. If so, we attempt to solve for an input which makes the value negative. We do this because negative values behave differently in signed and unsigned comparisons, and so they are likely to exhibit an error if one exists. All of this information is present in the trace without requiring access to the original program source code.

We discovered, however, that `gcc 4.1.2` inlines some calls to `memcpy` by transforming them to `rep movsb` instructions, even when the `-0` flag is not present. Furthermore, the Valgrind IR generated for the `rep movsb` instruction compares a decrementing counter variable to zero, instead of counting up and executing an unsigned comparison to the loop bound. As a result, on `gcc 4.1.2` a call to `memcpy` does not cause its length argument to be marked as unsigned. To deal with this problem, we implemented a simple heuristic to detect the IR generated for `rep movsb` and emit the appropriate con-

straint. We verified that this heuristic works on a small test case similar to Figure 3, generating a test input that caused a segmentation fault.

A key problem is storing all of the information required to carry out type inference without exhausting available memory. Because a trace may have several million instructions, memory usage is key to scaling type inference to long traces. Furthermore, our algorithm requires us to keep track of the types of all values in the program, unlike constraint generation, which need concern itself only with tainted values. An earlier version of our analysis created a special “type variable” for each value, then maintained a map from IR locations to type variables. Each type variable then mapped to a type. We found that in addition to being hard to maintain, this analysis often led to a number of live type variables that scaled linearly with the number of executed IR instructions. The result was that our analysis ran out of memory when attempting to play media files in the `mplayer` media player.

To solve this problem, we developed a garbage-collected data structure for tracking type information. To reduce memory consumption, we use a union-find data structure to partition integer values into equivalence classes where all values in an equivalence class are required to have the same type. We maintain one type for each union-find equivalence class; in our implementation type information is associated with the representative node for that equivalence class. Assignments force the source and target values to have the same types, which is implemented by merging their equivalence classes. Updating the type for a value can be done by updating its representative node’s type, with no need to explicitly update the types of all other variables in the equivalence class.

It turns out that this data structure is acyclic, due to the fact that VEX IR is in SSA form. Therefore, we use reference counting to garbage collect these nodes. In addition, we benefit from an additional property of the VEX IR: all values are either stored in memory, in registers, or in a temporary variable, and the lifetime of each temporary variable is implicitly limited to that of a single basic block. Therefore, we maintain a list of temporaries that are live in the current basic block; when we leave the basic block, the type information associated with all of those live temporaries can be deallocated. Consequently, the amount of memory needed for type inference at any point is proportional to the number of tainted (symbolic) variables that are live at that point—which is a significant improvement over the naive approach to type inference. The full version of this paper contains a more detailed specification of these algorithms³.

³<http://www.cs.berkeley.edu/~dmolnar/usenix09-full.pdf>

6 Triage and Reporting at Scale

Both SmartFuzz and zzuf can produce hundreds to thousands of test cases for a single test run. We designed and built a web service, *Metafuzz*, to manage the volume of tests. We describe some problems we found while building Metafuzz and techniques to overcoming these problems. Finally, we describe the user experience with Metafuzz and bug reporting.

6.1 Problems and Techniques

The Metafuzz architecture is as follows: first, a Test Machine generates new test cases for a program and runs them locally. The Test Machine then determines which test cases exhibit bugs and sends these test cases to Metafuzz. The Metafuzz web site displays these test cases to the User, along with information about what kind of bug was found in which target program. The User can pick test cases of interest and download them for further investigation. We now describe some of the problems we faced when designing Metafuzz, and our techniques for handling them. Section 7 reports our experiences with using Metafuzz to manage test cases and report bugs.

Problem: Each test run generated many test cases, too many to examine by hand.

Technique: We used Valgrind’s `memcheck` to automate the process of checking whether a particular test case causes the program to misbehave. `Memcheck` looks for memory leaks, use of uninitialized values, and memory safety errors such as writes to memory that was not allocated [26]. If `memcheck` reports an error, we save the test case. In addition, we looked for core dumps and non-zero program exit codes.

Problem: Even after filtering out the test cases that caused no errors, there were still many test cases that do cause errors.

Technique: The `metafuzz.com` front page is a HTML page listing all of the potential bug reports, showing all potential bug reports. Each test machine uploads information about test cases that trigger bugs to Metafuzz.

Problem: The machines used for testing had no long-term storage. Some of the test cases were too big to attach in e-mail or Bugzilla, making it difficult to share them with developers.

Technique: Test cases are uploaded directly to Metafuzz, providing each one with a stable URL. Each test case also includes the Valgrind output showing the Valgrind error, as well as the output of the program to `stdout` and `stderr`.

Problem: Some target projects change quickly. For example, we saw as many as four updates per day to the `mplayer` source code repository. Developers reject bug reports against “out of date” versions of the software.

Technique: We use the Amazon Elastic Compute Cloud (EC2) to automatically attempt to reproduce the bug

against the latest version of the target software. A button on the Metafuzz site spawns an Amazon EC2 instance that checks out the most recent version of the target software, builds it, and then attempts to reproduce the bug.

Problem: Software projects have specific reporting requirements that are tedious to implement by hand. For example, `mplayer` developers ask for a stack backtrace, disassembly, and register dump at the point of a crash.

Technique: Metafuzz automatically generates bug reports in the proper format from the failing test case. We added a button to the Metafuzz web site so that we can review the resulting bug report and then send it to the target software's bug tracker with a single click.

Problem: The same bug often manifests itself as many failing test cases. Reporting the same bug to developers many times wastes developer time.

Technique: We use the call stack to identify multiple instances of the same bug. Valgrind memcheck reports the call stack at each error site, as a sequence of instruction pointers. If debugging information is present, it also reports the associated filename and line number information in the source code.

Initially, we computed a *stack hash* as a hash of the sequence of instruction pointers in the backtrace. This has the benefit of not requiring debug information or symbols. Unfortunately, we found that a naive stack hash has several problems. First, it is sensitive to address space layout randomization (ASLR), because different runs of the same program may load the stack or dynamically linked libraries at different addresses, leading to different hash values for call stacks that are semantically the same. Second, even without ASLR, we found several cases where a single bug might be triggered at multiple call stacks that were similar but not identical. For example, a buggy function can be called in several different places in the code. Each call site then yields a different stack hash. Third, any slight change to the target software can change instruction pointers and thus cause the same bug to receive a different stack hash. While we do use the stack hash on the client to avoid uploading test cases for bugs that have been previously found, we found that we could not use stack hashes alone to determine if a bug report is novel or not.

To address these shortcomings, we developed a *fuzzy stack hash* that is forgiving of slight changes to the call stack. We use debug symbol information to identify the name of the function called, the line number in source code (excluding the last digit of the line number, to allow for slight changes in the code), and the name of the object file for each frame in the call stack. We then hash all of this information for the three functions at the top of the call stack.

The choice of the number of functions to hash determines the “fuzziness” of the hash. At one extreme, we

could hash all extant functions on the call stack. This would be similar to the classic stack hash and report many semantically same bugs in different buckets. On the other extreme, we could hash only the most recently called function. This fails in cases where two semantically different bugs both exhibit as a result of calling `memcpy` or some other utility function with bogus arguments. In this case, both call stacks would end with `memcpy` even though the bug is in the way the arguments are computed. We chose three functions as a trade-off between these extremes; we found this sufficient to stop further reports from the `mplayer` developers of duplicates in our initial experiences. Finding the best fuzzy stack hash is interesting future work; we note that the choice of bug bucketing technique may depend on the program under test.

While any fuzzy stack hash, including ours, may accidentally lump together two distinct bugs, we believe this is less serious than reporting duplicate bugs to developers. We added a post-processing step on the server that computes the fuzzy stack hash for test cases that have been uploaded to Metafuzz and uses it to coalesce duplicates into a single bug bucket.

Problem: Because Valgrind memcheck does not terminate the program after seeing an error, a single test case may give rise to dozens of Valgrind error reports. Two different test cases may share some Valgrind errors but not others.

Technique: First, we put a link on the Metafuzz site to a single test case for each bug bucket. Therefore, if two test cases share some Valgrind errors, we only use one test case for each of the errors in common. Second, when reporting bugs to developers, we highlight in the title the specific bugs on which to focus.

7 Results

7.1 Preliminary Experience

We used an earlier version of SmartFuzz and Metafuzz in a project carried out by a group of undergraduate students over the course of eight weeks in Summer 2008. When beginning the project, none of the students had any training in security or bug reporting. We provided a one-week course in software security. We introduced SmartFuzz, zzuf, and Metafuzz, then asked the students to generate test cases and report bugs to software developers. By the end of the eight weeks, the students generated over 1.2 million test cases, from which they reported over 90 bugs to software developers, principally to the `mplayer` project, of which 14 were fixed. For further details, we refer to their presentation [1].

7.2 Experiment Setup

Test Programs. Our target programs were `mplayer` version SVN-r28403-4.1.2, `ffmpeg` version

	mplayer	ffmpeg	exiv2	gzip	bzip2	convert		Queries	Test Cases	Bugs
Coverage	2599	14535	1629	5906	12606	388	Coverage	588068	31121	19
ConversionNot32	0	3787	0	0	0	0	ConversionNot32	4586	0	0
Conversion32to8	1	26	740	2	10	116	Conversion32to8	1915	1377	3
Conversion32to16	0	16004	0	0	0	0	Conversion32to16	16073	67	4
Conversion16to32	0	121	0	0	0	0	Conversion16to32	206	0	0
SignedOverflow	1544	37803	5941	24825	9109	49	SignedOverflow	167110	0	0
SignedUnderflow	3	4003	48	1647	2840	0	SignedUnderflow	20198	21	3
UnsignedOverflow	1544	36945	4957	24825	9104	35	UnsignedOverflow	164155	9280	3
UnsignedUnderflow	0	0	0	0	0	0	MallocArg	30	0	0
MallocArg	0	24	0	0	0	0	SignedUnsigned	125509	6949	5
SignedUnsigned	2568	21064	799	7883	17065	49				

Figure 5: The number of each type of query for each test program after a single 24-hour run.

SVN-r16903, exiv2 version SVN-r1735, gzip version 1.3.12, bzip2 version 1.0.5, and ImageMagick convert version 6.4.8 – 10, which are all widely used media and compression programs. Table 4 shows information on the size of each test program. Our test programs are large, both in terms of source lines of code and trace lengths. The percentage of the trace that is symbolic, however, is small.

Test Platform. Our experiments were run on the Amazon Elastic Compute Cloud (EC2), employing a “small” and a “large” instance image with SmartFuzz, zzuf, and all our test programs pre-installed. At this writing, an EC2 small instance has 1.7 GB of RAM and a single-core virtual CPU with performance roughly equivalent to a 1GHz 2007 Intel Xeon. An EC2 large instance has 7 GB of RAM and a dual-core virtual CPU, with each core having performance roughly equivalent to a 1 GHz Xeon.

We ran all mplayer runs and ffmpeg runs on EC2 large instances, and we ran all other test runs with EC2 small instances. We spot-checked each run to ensure that instances successfully held all working data in memory during symbolic execution and triage without swapping to disk, which would incur a significant performance penalty. For each target program we ran SmartFuzz and zzuf with three seed files, for 24 hours per program per seed file. Our experiments took 288 large machine-hours and 576 small machine-hours, which at current EC2 prices of \$0.10 per hour for small instances and \$0.40 per hour for large instances cost \$172.80.

Query Types. SmartFuzz queries our solver with the following types of queries: Coverage, ConversionNot32, Conversion32to8, Conversion32to16, SignedOverflow, UnsignedOverflow, SignedUnderflow, UnsignedUnderflow, MallocArg, and SignedUnsigned. Coverage queries refer to queries created as part of the generational search by flipping path conditions. The others are *bug-seeking queries* that attempt to synthesize inputs leading to specific kinds of bugs. Here MallocArg refers to a

Figure 6: The number of bugs found, by query type, over all test runs. The fourth column shows the number of distinct bugs found from test cases produced by the given type of query, as classified using our fuzzy stack hash.

set of bug-seeking queries that attempt to force inputs to known memory allocation functions to be negative, yielding an implicit conversion to a large unsigned integer, or force the input to be small.

Experience Reporting to Developers. Our original strategy was to report all distinct bugs to developers and let them judge which ones to fix. The mplayer developers gave us feedback on this strategy. They wanted to focus on fixing the most serious bugs, so they preferred seeing reports only for out-of-bounds writes and double free errors. In contrast, they were not as interested in out-of-bound reads, even if the resulting read caused a segmentation fault. This helped us prioritize bugs for reporting.

7.3 Bug Statistics

Integer Bug-Seeking Queries Yield Bugs. Figure 6 reports the number of each type of query to the constraint solver over all test runs. For each type of query, we report the number of test files generated and the number of distinct bugs, as measured by our fuzzy stack hash. Some bugs may be revealed by multiple different kinds of queries, so there may be overlap between the bug counts in two different rows of Figure 6.

The table shows that our dynamic test generation methods for integer bugs succeed in finding bugs in our test programs. Furthermore, the queries for signed/unsigned bugs found the most distinct bugs out of all bug-seeking queries. This shows that our novel method for detecting signed/unsigned bugs (Section 5) is effective at finding bugs.

SmartFuzz Finds More Bugs Than zzuf, on mplayer. For mplayer, SmartFuzz generated 10,661 test cases over all test runs, while zzuf generated 11,297 test cases; SmartFuzz found 22 bugs while zzuf found 13. Therefore, in terms of number of bugs, SmartFuzz outperformed zzuf for testing mplayer. Another surprising result here is that SmartFuzz generated nearly as many test cases as zzuf, despite the additional overhead for

	SLOC	seedfile type and size	Branches	x86 instrs	IRStmts	asserts	queries
mplayer	723468	MP3 (159000 bytes)	20647045	159500373	810829992	1960	36
ffmpeg	304990	AVI (980002 bytes)	4147710	19539096	115036155	4778690	462346
exiv2	57080	JPG (22844 bytes)	809806	6185985	32460806	81450	1006
gzip	140036	TAR.GZ (14763 bytes)	24782	161118	880386	95960	13309
bzip	26095	TAR.BZ2 (618620 bytes)	107396936	746219573	4185066021	1787053	314914
ImageMagick	300896	PNG (25385 bytes)	98993374	478474232	2802603384	583	81

Figure 4: The size of our test programs. We report the source lines of code for each test program and the size of one of our seed files, as measured by David A. Wheeler’s `sloccount`. Then we run the test program on that seed file and report the total number of branches, x86 instructions, Valgrind IR statements, STP assert statements, and STP query statements for that run. We ran symbolic execution for a maximum of 12 hours, which was sufficient for all programs except `mplayer`, which terminated during symbolic execution.

symbolic execution and constraint solving. This shows the effect of the generational search and the choice of memory model; we leverage a single expensive symbolic execution and fast solver queries to generate many test cases. At the same time, we note that `zzuf` found a serious `InvalidWrite` bug, while `SmartFuzz` did not.

A previous version of our infrastructure had problems with test cases that caused the target program to run forever, causing the search to stall. Therefore, we introduced a timeout, so that after 300 CPU seconds, the target program is killed. We manually examined the output of `memcheck` from all killed programs to determine whether such test cases represent errors. For `gzip` we discovered that `SmartFuzz` created six such test cases, which account for the two out-of-bounds read (`InvalidRead`) errors we report; `zzuf` did not find any hanging test cases for `gzip`. We found no other hanging test cases in our other test runs.

Different Bugs Found by `SmartFuzz` and `zzuf`. We ran the same target programs with the same seed files using `zzuf`. Figure 7 shows bugs found by each type of fuzzer. With respect to each tool, `SmartFuzz` found 37 total distinct bugs and `zzuf` found 59 distinct bugs. We found some overlap between bugs as well: 19 bugs were found by both fuzz testing tools, for a total of 77 distinct bugs. This shows that while there is overlap between the two tools, `SmartFuzz` finds bugs that `zzuf` does not and vice versa. Therefore, it makes sense to try both tools when testing software.

Note that we did not find any bugs for `bzip2` with either fuzzer, so neither tool was effective on this program. This shows that fuzzing is not always effective at finding bugs, especially with a program that has already seen attention for security vulnerabilities. We also note that `SmartFuzz` found `InvalidRead` errors in `gzip` while `zzuf` found no bugs in this program. Therefore `gzip` is a case where `SmartFuzz`’s directed testing is able to trigger a bug, but purely random testing is not.

Block Coverage. We measured the number of basic blocks in the program visited by the execution of the

seed file, then measured how many new basic blocks were visited during the test run. We discovered `zzuf` added a higher percentage of new blocks than `SmartFuzz` in 13 of the test runs, while `SmartFuzz` added a higher percentage of new blocks in 4 of the test runs (the `SmartFuzz convert-2` test run terminated prematurely.) Table 8 shows the initial basic blocks, the number of blocks added, and the percentage added for each fuzzer. We see that the effectiveness of `SmartFuzz` varies by program; for `convert` it is particularly effective, finding many more new basic blocks than `zzuf`.

Contrasting `SmartFuzz` and `zzuf` Performance. Despite the limitations of random testing, the blackbox fuzz testing tool `zzuf` found bugs in four out of our six test programs. In three of our test programs, `zzuf` found more bugs than `SmartFuzz`. Furthermore, `zzuf` found the most serious `InvalidWrite` errors, while `SmartFuzz` did not. These results seem surprising, because `SmartFuzz` exercises directed testing based on program behavior while `zzuf` is a purely blackbox tool. We would expect that `SmartFuzz` should find all the bugs found by `zzuf`, given an unbounded amount of time to run both test generation methods.

In practice, however, the time which we run the methods is limited, and so the question is which bugs are discovered first by each method. We have identified possible reasons for this behavior, based on examining the test cases and Valgrind errors generated by both tools⁴. We now list and briefly explain these reasons.

Header parsing errors. The errors we observed are often in code that parses the header of a file format. For example, we noticed bugs in functions of `mplayer` that parse MP3 headers. These errors can be triggered by simply placing “wrong” values in data fields of header files. As a result, these errors do not require complicated and unlikely predicates to be true before reaching buggy code, and so the errors can be reached without needing the full power of dynamic test generation. Similarly, code cov-

⁴We have placed representative test cases from each method at <http://www.metafuzz.com/example-testcases.tgz>

	mplayer		ffmpeg		exiv2		gzip		convert	
SyscallParam	4	3	2	3	0	0	0	0	0	0
UninitCondition	13	1	1	8	0	0	0	0	3	8
UninitValue	0	3	0	3	0	0	0	0	0	2
Overlap	0	0	0	1	0	0	0	0	0	0
Leak_DefinitelyLost	2	2	2	4	0	0	0	0	0	0
Leak_PossiblyLost	2	1	0	2	0	1	0	0	0	0
InvalidRead	1	2	0	4	4	6	2	0	1	1
InvalidWrite	0	1	0	3	0	0	0	0	0	0
Total	22	13	5	28	4	7	2	0	4	11
Cost per bug	\$1.30	\$2.16	\$5.76	\$1.03	\$1.80	\$1.03	\$3.60	NA	\$1.20	\$0.65

Figure 7: The number of bugs, after fuzzy stack hashing, found by SmartFuzz (the number on the left in each column) and zzuf (the number on the right). We also report the cost per bug, assuming \$0.10 per small compute-hour, \$0.40 per large compute-hour, and 3 runs of 24 hours each per target for each tool.

erage may be affected by the style of program used for testing.

Difference in number of bytes changed. The two different methods change a vastly different number of bytes from the original seed file to create each new test case. SmartFuzz changes exactly those bytes that must change to force program execution down a single new path or to violate a single property. Below we show that in our programs there are only a small number of constraints on the input to solve, so a SmartFuzz test case differs from its seed file by only a small number of bytes. In contrast, zzuf changes a fixed fraction of the input bytes to random other bytes; in our experiments, we left this at the default value of 0.004.

The large number of changes in a single fuzzed test case means that for header parsing or other code that looks at small chunks of the file independently, a single zzuf test case will exercise many different code paths with fuzzed chunks of the input file. Each path which originates from parsing a fuzzed chunk of the input file is a potential bug. Furthermore, because Valgrind memcheck does not necessarily terminate the program’s execution when a memory safety error occurs, one such file may yield multiple bugs and corresponding bug buckets.

In contrast, the SmartFuzz test cases usually change only one chunk and so explore only one “fuzzed” path through such code. Our code coverage metric, unfortunately, is not precise enough to capture this difference because we measured block coverage instead of path coverage. This means once a set of code blocks has been covered, a testing method receives no further credit for additional paths through those same code blocks.

Small number of generations reached by SmartFuzz. Our 24 hour experiments with SmartFuzz tended to reach a small number of generations. While previous work on whitebox fuzz testing shows that most bugs are found in the early generations [14], this feeds into the previous issue because the number of differences between the fuzzed file and the original file is proportional to the

generation number of the file. We are exploring longer-running SmartFuzz experiments of a week or more to address this issue.

Loop behavior in SmartFuzz. Finally, SmartFuzz deals with loops by looking at the unrolled loop in the dynamic program trace, then attempting to generate test cases for each symbolic if statement in the unrolled loop. This strategy is not likely to create new test cases that cause the loop to execute for vastly more iterations than seen in the trace. By contrast, the zzuf case may get lucky by assigning a random and large value to a byte sequence in the file that controls the loop behavior. On the other hand, when we looked at the gzip bug found by SmartFuzz and not by zzuf, we discovered that it appears to be due to an infinite loop in the inflate_dynamic routine of gzip.

7.4 SmartFuzz Statistics

Integer Bug Queries Vary By Program. Table 5 shows the number of solver queries of each type for one of our 24-hour test runs. We see that the type of queries varies from one program to another. We also see that for bzip2 and mplayer, queries generated by type inference for signed/unsigned errors account for a large fraction of all queries to the constraint solver. This results from our choice to eagerly generate new test cases early in the program; because there are many potential integer bugs in these two programs, our symbolic traces have many integer bug-seeking queries. Our design choice of using an independent tool such as memcheck to filter the resulting test cases means we can tolerate such a large number of queries because they require little human oversight.

Time Spent In Each Task Varies By Program. Figure 9 shows the percentage of time spent in symbolic execution, coverage, triage, and recording for each run of our experiment. We also report an “Other” category, which includes the time spent in the constraint solver. This shows us where we can obtain gains through further optimization. The amount of time spent in each task depends greatly on the seed file, as well as on the

Test run	Initial basic blocks		Blocks added by tests		Ratio of prior two columns	
	SmartFuzz	zzuf	SmartFuzz	zzuf	SmartFuzz	zzuf
mplayer-1	7819	7823	5509	326	70%	4%
mplayer-2	11375	11376	908	1395	7%	12%
mplayer-3	11093	11096	102	2472	0.9%	22%
ffmpeg-1	6470	6470	592	20036	9.14%	310%
ffmpeg-2	6427	6427	677	2210	10.53%	34.3%
ffmpeg-3	6112	611	97	538	1.58%	8.8%
convert-1	8028	8246	2187	20	27%	0.24%
convert-2	8040	8258	2392	6	29%	0.073%
convert-3	NA	10715	NA	1846	NA	17.2%
exiv2-1	9819	9816	2934	3560	29.9%	36.3%
exiv2-2	9811	9807	2783	3345	28.3%	34.1%
exiv2-3	9814	9810	2816	3561	28.7%	36.3%
gzip-1	2088	2088	252	334	12%	16%
gzip-2	2169	2169	259	275	11.9%	12.7%
gzip-3	2124	2124	266	316	12%	15%
bzip2-1	2779	2778	123	209	4.4%	7.5%
bzip2-2	2777	2778	125	237	4.5%	8.5%
bzip2-3	2823	2822	115	114	4.1%	4.0%

Figure 8: Coverage metrics: the initial number of basic blocks, before testing; the number of blocks added during testing; and the percentage of blocks added.

	Total	SymExec	Coverage	Triage	Record	Other
gzip-1	206522s	0.1%	0.06%	0.70%	17.6%	81.6%
gzip-2	208999s	0.81%	0.005%	0.70%	17.59%	80.89%
gzip-3	209128s	1.09%	0.0024%	0.68%	17.8%	80.4%
bzip2-1	208977s	0.28%	0.335%	1.47%	14.0%	83.915%
bzip2-2	208849s	0.185%	0.283%	1.25%	14.0%	84.32%
bzip2-3	162825s	25.55%	0.78%	31.09%	3.09%	39.5%
mplayer-1	131465s	14.2%	5.6%	22.95%	4.7%	52.57%
mplayer-2	131524s	15.65%	5.53%	22.95%	25.20%	30.66%
mplayer-3	49974s	77.31%	0.558%	1.467%	10.96%	9.7%
ffmpeg-1	73981s	2.565%	0.579%	4.67%	70.29%	21.89%
ffmpeg-2	131600s	36.138%	1.729%	9.75%	11.56%	40.8%
ffmpeg-3	24255s	96.31%	0.1278%	0.833%	0.878%	1.8429%
convert-1	14917s	70.17%	2.36%	24.13%	2.43%	0.91%
convert-2	97519s	66.91%	1.89%	28.14%	2.18%	0.89%
exiv2-1	49541s	3.62%	10.62%	71.29%	9.18%	5.28%
exiv2-2	69415s	3.85%	12.25%	65.64%	12.48%	5.78%
exiv2-3	154334s	1.15%	1.41%	3.50%	8.12%	85.81%

Figure 9: The percentage of time spent in each of the phases of SmartFuzz. The second column reports the total wall-clock time, in seconds, for the run; the remaining columns are a percentage of this total. The “other” column includes the time spent solving STP queries.

target program. For example, the first run of mplayer, which used a mp3 seedfile, spent 98.57% of total time in symbolic execution, and only 0.33% in coverage, and 0.72% in triage. In contrast, the second run of mplayer, which used a mp4 seedfile, spent only 14.77% of time in symbolic execution, but 10.23% of time in coverage, and 40.82% in triage. We see that the speed of symbolic execution and of triage is the major bottleneck for several of our test runs, however. This shows that future work should focus on improving these two areas.

7.5 Solver Statistics

Related Constraint Optimization Varies By Program.

We measured the size of all queries to the constraint solver, both before and after applying the related constraint optimization described in Section 4. Figure 10 shows the average size for queries from each test program, taken over all queries in all test runs with that program. We see that while the optimization is effective in all cases, its average effectiveness varies greatly from one test program to another. This shows that different programs vary greatly in how many input bytes influence each query.

The Majority of Queries Are Fast. Figure 10 shows the empirical cumulative distribution function of STP solver times over all our test runs. For about 70% of the test cases, the solver takes at most one second. The maximum solver time was about 10.89 seconds. These results reflect our choice of memory model and the effectiveness of the related constraint optimization. Because of these, the queries to the solver consist only of operations over bitvectors (with no array constraints), and most of the sets of constraints sent to the solver are small, yielding fast solver performance.

8 Conclusion

We described new methods for finding integer bugs in dynamic test generation, and we implemented these methods in SmartFuzz, a new dynamic test generation tool. We then reported on our experiences building the web site metafuzz.com and using it to manage test case generation at scale. In particular, we found that SmartFuzz finds bugs not found by zzuf and vice versa, showing that a comprehensive testing strategy should use both

	average before	average after	ratio (before/after)
mplayer	292524	33092	8.84
ffmpeg	350846	83086	4.22
exiv2	81807	10696	7.65
gzip	348027	199336	1.75
bzip2	278980	162159	1.72
convert	2119	1057	2.00

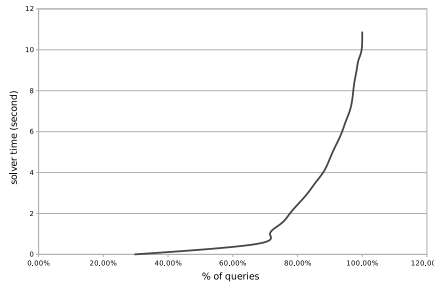


Figure 10: On the left, average query size before and after related constraint optimization for each test program. On the right, an empirical CDF of solver times.

white-box and black-box test generation tools.

Furthermore, we showed that our methods can find integer bugs without the false positives inherent to static analysis or runtime checking approaches, and we showed that our methods scale to commodity Linux media playing software. The Metafuzz web site is live, and we have released our code to allow others to use our work.

9 Acknowledgments

We thank Cristian Cadar, Daniel Dunbar, Dawson Engler, Patrice Godefroid, Michael Levin, and Paul Twohey for discussions about their respective systems and dynamic test generation. We thank Paul Twohey for helpful tips on the engineering of test machines. We thank Chris Karlof for reading a draft of our paper on short notice. We thank the SUPERB TRUST 2008 team for their work with Metafuzz and SmartFuzz during Summer 2008. We thank Li-Wen Hsu and Alex Fabrikant for their help with the metafuzz.com web site, and Sushant Shankar, Shiuan-Tzuo Shen, and Mark Winterrowd for their comments. We thank Erinn Clark, Charlie Miller, Prateek Saxena, Dawn Song, the Berkeley BitBlaze group, and the anonymous Oakland referees for feedback on earlier drafts of this work. This work was generously supported by funding from DARPA and by NSF grants CCF-0430585 and CCF-0424422.

References

[1] ASLANI, M., CHUNG, N., DOHERTY, J., STOCKMAN, N., AND QUACH, W. Comparison of blackbox and whitebox fuzzers in finding software bugs, November 2008. TRUST Retreat Presentation.

[2] BLEXIM. Basic integer overflows. *Phrack 0x0b* (2002).

[3] BLEXIM. Basic integer overflows. *Phrack 0x0b, 0x3c* (2002). <http://www.phrack.org/archives/60/p60-0x0a.txt>.

[4] BRUMLEY, D., CHIEH, T., JOHNSON, R., LIN, H., AND SONG, D. RICH: Automatically protecting against integer-based vulnerabilities. In *NDSS (Symp. on Network and Distributed System Security)* (2007).

[5] BRUMLEY, D., NEWSOME, J., SONG, D., WANG, H., AND JHA, S. Towards automatic generation of vulnerability-based signatures. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy* (2006).

[6] CADAR, C., DUNBAR, D., AND ENGLER, D. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of OSDI 2008* (2008).

[7] CADAR, C., AND ENGLER, D. EGT: Execution generated testing. In *SPIN* (2005).

[8] CADAR, C., GANESH, V., PAWLOWSKI, P. M., DILL, D. L., AND ENGLER, D. R. EXE: Automatically Generating Inputs of Death. In *ACM CCS* (2006).

[9] CHEN, K., AND WAGNER, D. Large-scale analysis of format string vulnerabilities in debian linux. In *PLAS - Programming Languages and Analysis for Security* (2007). <http://www.cs.berkeley.edu/~daw/papers/fmtstr-plas07.pdf>.

[10] CORPORATION, M. Vulnerability Type Distributions in CVE, May 2007. <http://cve.mitre.org/docs/vuln-trends/index.html>.

[11] DEMOTT, J. The evolving art of fuzzing. In *DEF CON 14* (2006). http://www.appliedsec.com/files/The_Evolving_Art_of_Fuzzing.odp.

[12] GANESH, V., AND DILL, D. STP: A decision procedure for bitvectors and arrays. *CAV 2007*, 2007. <http://theory.stanford.edu/~vganesh/stp.html>.

[13] GODEFROID, P., KLARLUND, N., AND SEN, K. DART: Directed Automated Random Testing. In *Proceedings of PLDI'2005 (ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation)* (Chicago, June 2005), pp. 213–223.

- [14] GODEFROID, P., LEVIN, M., AND MOLNAR, D. Automated Whitebox Fuzz Testing. In *Proceedings of NDSS'2008 (Network and Distributed Systems Security)* (San Diego, February 2008). http://research.microsoft.com/users/pg/public_psfiles/ndss2008.pdf.
- [15] GODEFROID, P., LEVIN, M. Y., AND MOLNAR, D. Active Property Checking. Tech. rep., Microsoft, 2007. MSR-TR-2007-91.
- [16] HOCEVAR, S. zzuf, 2007. <http://caca.zoy.org/wiki/zzuf>.
- [17] LANZI, A., MARTIGNONI, L., MONGA, M., AND PALEARI, R. A smart fuzzer for x86 executables. In *Software Engineering for Secure Systems, 2007. SESS '07: ICSE Workshops 2007* (2007). <http://idea.sec.dico.unimi.it/~roberto/pubs/sess07.pdf>.
- [18] LARSON, E., AND AUSTIN, T. High Coverage Detection of Input-Related Security Faults. In *Proceedings of 12th USENIX Security Symposium* (Washington D.C., August 2003).
- [19] LEBLANC, D. Safeint 3.0.11, 2008. <http://www.codeplex.com/SafeInt>.
- [20] LMH. Month of kernel bugs, November 2006. <http://projects.info-pull.com/mokb/>.
- [21] MICROSOFT CORPORATION. Prefast, 2008.
- [22] MILLER, B. P., FREDRIKSEN, L., AND SO, B. An empirical study of the reliability of UNIX utilities. *Communications of the Association for Computing Machinery* 33, 12 (1990), 32–44.
- [23] MOORE, H. Month of browser bugs, July 2006. <http://browserfun.blogspot.com/>.
- [24] NETHERCOTE, N., AND SEWARD, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI - Programming Language Design and Implementation* (2007).
- [25] O'CALLAHAN, R. Chronicle-recorder, 2008. <http://code.google.com/p/chronicle-recorder/>.
- [26] SEWARD, J., AND NETHERCOTE, N. Using valgrind to detect undefined memory errors with bit precision. In *Proceedings of the USENIX Annual Technical Conference* (2005). <http://www.valgrind.org/docs/memcheck2005.pdf>.
- [27] VUAGNOUX, M. Autodafe: An act of software torture. In *22nd Chaos Communications Congress, Berlin, Germany* (2005). autodafe.sourceforge.net.
- [28] WANG, T., WEI, T., LIN, Z., AND ZOU, W. Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *Network Distributed Security Symposium (NDSS)* (2009).

Memory Safety for Low-Level Software/Hardware Interactions

John Criswell
University of Illinois
criswell@uiuc.edu

Nicolas Geoffray
Université Pierre et Marie Curie
INRIA/Regal
nicolas.geoffray@lip6.fr

Vikram Adve
University of Illinois
vadve@uiuc.edu

Abstract

Systems that enforce memory safety for today's operating system kernels and other system software do not account for the behavior of low-level software/hardware interactions such as memory-mapped I/O, MMU configuration, and context switching. Bugs in such low-level interactions can lead to violations of the memory safety guarantees provided by a safe execution environment and can lead to exploitable vulnerabilities in system software. In this work, we present a set of program analysis and run-time instrumentation techniques that ensure that errors in these low-level operations do not violate the assumptions made by a safety checking system. Our design introduces a small set of abstractions and interfaces for manipulating processor state, kernel stacks, memory mapped I/O objects, MMU mappings, and self modifying code to achieve this goal, without moving resource allocation and management decisions out of the kernel. We have added these techniques to a compiler-based virtual machine called Secure Virtual Architecture (SVA), to which the standard Linux kernel has been ported previously. Our design changes to SVA required only an additional 100 lines of code to be changed in this kernel. Our experimental results show that our techniques prevent reported memory safety violations due to low-level Linux operations and that *these violations are not prevented by SVA without our techniques*. Moreover, the new techniques in this paper introduce very little overhead over and above the existing overheads of SVA. Taken together, these results indicate that it is clearly worthwhile to add these techniques to an existing memory safety system.

1 Introduction

Most modern system software, including commodity operating systems and virtual machine monitors, are vulnerable to a wide range of security attacks because they are written in unsafe languages like C and C++. In

fact, there has been an increase in recent years of attack methods against the operating system (OS) kernel. There are reported vulnerabilities for nearly all commodity OS kernels (e.g., [2, 28, 43]). One recent project went so far as to present one OS kernel bug every day for a month for several different open source and commercial kernels [26] (several of these bugs are exploitable vulnerabilities). Preventing these kinds of attacks *requires protecting the core kernel and not just device drivers*: many of the vulnerabilities are in core kernel components [19, 40, 41, 43, 46].

To counter these threats, there is a growing body of work on using language and compiler techniques to enforce *memory safety* (defined in Section 2) for OS code. These include new OS designs based on safe languages [4, 18, 22, 33], compiler techniques to enforce memory safety for commodity OSs in unsafe languages [10], and instrumentation techniques to isolate a kernel from extensions such as device drivers [45, 47, 51]. We use the term “*safe execution environment*” (again defined in Section 2) to refer to the guarantees provided by a system that enforces memory safety for operating system code. Singularity, SPIN, JX, JavaOS, SafeDrive, and SVA are examples of systems that enforce a safe execution environment.

Unfortunately, all these memory safety techniques (even implementations of safe programming languages) make assumptions that are routinely violated by low-level interactions between an OS kernel and hardware. Such assumptions include a static, one-to-one mapping between virtual and physical memory, an idealized processor whose state is modified only via visible program instructions, I/O operations that cannot overwrite standard memory objects except input I/O targets, and a protected stack modifiable only via load/store operations to local variables. For example, when performing type checking on a method, a safe language like Java or Modula-3 or compiler techniques like those in SVA assume that pointer values are only defined via visible pro-

gram operations. In a kernel, however, a *buggy* kernel operation might overwrite program state while it is off-processor and that state might later be swapped in between the definition and the use of the pointer value, a *buggy* MMU mapping might remap the underlying physical memory to a different virtual page holding data of a different type, or a *buggy* I/O operation might bring corrupt pointer values into memory.

In fact, as described in Section 7.1, we have injected bugs into the Linux kernel ported to SVA that are capable of *disabling the safety checks that prevented 3 of the 4 exploits* in the experiments reported in the original SVA work [10]: the bugs modify the metadata used to track array bounds and thus allow buffer overruns to go undetected. Similar vulnerabilities can be introduced with other bugs in low-level operations. For example, there are reported MMU bugs [3, 39, 42] in previous versions of the Linux kernel that are logical errors in the MMU configuration and could lead to kernel exploits.

A particularly nasty and very recent example is an insidious error in the Linux 2.6 kernel (not a device driver) that led to severe (and sometimes permanent) corruption of the e1000e network card [9]. The kernel was overwriting I/O device memory with the x86 `cmpxchg` instruction, which led to corrupting the hardware. This bug was caused by a write through a dangling pointer to I/O device memory. This bug took weeks of debugging by multiple teams to isolate. A strong memory safety system should prevent or constrain such behavior, either of which would have prevented the bug.

All these problems can, in theory, be prevented by moving some of the kernel-hardware interactions into a virtual machine (VM) and providing a high-level interface for the OS to invoke those operations safely. If an OS is *co-designed* with a virtual machine implementing the underlying language, e.g., as in JX [18], then eliminating such operations from the kernel could be feasible. For commodity operating systems such as Linux, Mac OS X, and Windows, however, reorganizing the OS in such a way may be difficult or impossible, requiring, at a minimum, substantial changes to the OS design. For example, in the case of SVA, moving kernel-hardware interactions into the SVA VM would require extensive changes to any commodity system ported to SVA.

Virtual machine monitors (VMMs) such as VMWare or Xen [16] do not solve this problem. They provide sufficiently strong guarantees to enforce isolation and fair resource sharing between different OS instances (i.e., different “domains”) but do not enforce memory safety *within* a single instance of an OS. For example, a VMM prevents one OS instance from modifying memory mappings for a different instance but does not protect an OS instance from a bug that maps multiple pages of its own to the same physical page, thus violating necessary as-

sumptions used to enforce memory safety. In fact, a VMM would not solve any of the reported real-world problems listed above.

In this paper, we present a set of novel techniques to prevent low-level kernel-hardware interactions from violating memory safety in an OS executing in a safe execution environment. There are two key aspects to our approach: (1) we define carefully a set of abstractions (an API) between the kernel and the hardware that enables a lightweight run-time checker to protect hardware resources and their behaviors; and (2) we leverage the existing safety checking mechanisms of the safe execution environment to *optimize* the extra checks that are needed for this monitoring. Some examples of the key resources that are protected by our API include processor state in CPU registers; processor state saved in memory on context-switches, interrupts, or system calls; kernel stacks; memory-mapped I/O locations; and MMU configurations. Our design also permits limited versions of self-modifying code that should suffice for most kernel uses of the feature. Most importantly, our design provides these assurances while leaving essentially all the *logical control* over hardware behavior in the hands of the kernel, i.e., no policy decisions or complex mechanisms are taken out of the kernel. Although we focus on preserving memory safety for commodity operating systems, these principles would enable any OS to reduce the likelihood and severity of failures due to bugs in low-level software-hardware interactions.

We have incorporated these techniques in the SVA prototype and correspondingly modified the Linux 2.4.22 kernel previously ported to SVA [10]. Our new techniques required a significant redesign of SVA-OS, which is the API provided by SVA to a kernel for controlling hardware and using privileged hardware operations. The changes to the Linux kernel were generally simple changes to use the new SVA-OS API, even though the new API provides much more powerful protection for the entire kernel. We had to change only about 100 lines in the SVA kernel to conform to the new SVA-OS API.

We have evaluated the ability of our system to prevent kernel bugs due to kernel-hardware interactions, both with real reported bugs and injected bugs. Our system prevents two MMU bugs in Linux 2.4.22 for which exploit code is available. Both bugs crash the kernel when run under the original SVA. Moreover, as explained in Section 7.1, we would also prevent the e1000e bug in Linux 2.6 if that kernel is run on our system. Finally, the system successfully prevents all the low-level kernel-hardware interaction errors we have tried to inject.

We also evaluated the performance overheads for two servers and three desktop applications (two of which perform substantial I/O). Compared with the original SVA, the new techniques in this paper add very low or negli-

ble overheads. Combined with the ability to prevent real-world exploits that would be missed otherwise, it clearly seems worthwhile to add these techniques to an existing memory safety system.

To summarize, the key contributions of this work are:

- We have presented novel mechanisms to ensure that low-level kernel-hardware interactions (e.g., context switching, thread creation, MMU changes, and I/O operations) do not violate assumptions used to enforce a safe execution environment.
- We have prototyped these techniques and shown that they can be used to enforce the assumptions made by a memory safety checker for a commodity kernel such as Linux. To our knowledge, no previous safety enforcement technique provides such guarantees to commodity system software.
- We have evaluated this system experimentally and shown that it is effective at preventing exploits in the above operations in Linux while incurring little overhead over and above the overhead of the underlying safe execution environment of SVA.

2 Breaking Memory Safety with Low-Level Kernel Operations

Informally, a *program is type-safe* if all operations in the program respect the types of their operands. For the purposes of this work, we say a *program is memory safe* if every memory access uses a previously initialized pointer variable; accesses the same object to which the pointer pointed initially;¹ and the object has not been deallocated. Memory safety is necessary for type safety (conversely, type safety implies memory safety) because dereferencing an uninitialized pointer, accessing the target object out of bounds, or dereferencing a dangling pointer to a freed object, can all cause accesses to unpredictable values and hence allow illegal operations on those values.

A safe programming language guarantees type safety and memory safety for all legal programs [34]; these guarantees also imply a *sound operational semantics* for programs in the language. Language implementations enforce these guarantees through a combination of compile-time type checking, automatic memory management (e.g., garbage collection or region-based memory management) to prevent dangling pointer references, and run-time checks such as array bounds checks and null pointer checks.

Four recent compiler-based systems for C, namely, CCured [30], SafeDrive [51], SAFECode [15], and

SVA [10] enforce similar, but weaker, guarantees for C code. Their guarantees are weaker in two ways: (a) they provide type safety for only a subset of objects, and (b) three of the four systems — SafeDrive, SAFECode and SVA — permit dangling pointer references (use-after-free) to avoid the need for garbage collection. Unlike SafeDrive, however, SAFECode and SVA *guarantee* that dangling pointer references do not invalidate any of the other safety properties, i.e., partial type safety, memory safety, or a sound operational semantics [14, 15]. We refer to all these systems – safe languages or safety checking compilers – as providing a *safe execution environment*.

All of the above systems make some fundamental assumptions regarding the run-time environment in enforcing their safety guarantees. In particular, these systems assume that the code segment is static; control flow can only be altered through explicit branch instructions, call instructions, and visible signal handling; and that data is stored either in a flat, unchanging address space or in processor registers. Furthermore, data can only be read and written by direct loads and stores to memory or direct changes to processor registers.

Low-level system code routinely violates these assumptions. Operating system kernels, virtual machine monitors, language virtual machines such as a JVM or CLR, and user-level thread libraries often perform operations such as context switching, direct stack manipulation, memory mapped I/O, and MMU configuration, that violate these assumptions. More importantly, as explained in the rest of this section, perfectly *type-safe* code can violate many of these assumptions (through logical errors), i.e., such errors will not be prevented by the language in the first place. This is unacceptable for safe language implementations and, at least, undesirable for system software because these violations can compromise safety and soundness and thus permit the vulnerabilities a safe language was designed to prevent, such as buffer overflows or the creation of illegal pointer values.

There are, in fact, a small number of root causes (or categories of root causes) of all these violations. This section enumerates these root causes, and the next section describes the design principles by which these root causes can be eliminated. We assume throughout this discussion that a safety checker (through some combination of static and run-time checking) enforces the language-level safety guarantees of a safe execution environment, described above, for the kernel.² This allows us to assume that the run-time checker itself is secure, and that static analysis can be used soundly on kernel code [15]. Our goal is to ensure the integrity of the *as-*

¹Note that we permit a pointer to “leave” its target object and later return, as long as it is not accessed while it is out of bounds [32].

²This work focuses on enforcing memory safety for the kernel. The same techniques could be applied to protect user-space threads from these violations.

assumptions made by this safety checker. We refer to the extensions that enforce these assumptions as a *verifier*.

Briefly, the fundamental categories of violations are:

- corrupting processor state when held in registers or memory;
- corrupting stack values for kernel threads;
- corrupting memory mapped I/O locations;
- corrupting code pages in memory;
- other violations that can corrupt arbitrary memory locations, including those listed above.

Unlike the last category, the first four above are errors that are specific to individual categories of memory.

2.1 Corrupting Processor State

Corrupting processor state can corrupt both data and control flow. The verifier must first ensure that processor state cannot be corrupted while on the processor itself, i.e., preventing arbitrary changes to processor registers. In addition, however, standard kernels save processor state (i.e., data and control registers) in memory where it is accessible by standard (even type-safe) load and store instructions. Any (buggy) code that modifies this state before restoring the state to the processor can alter control flow (the program counter, stack pointer, return address register, or condition code registers) or data values. In safe systems that permit dangling pointer references, processor state can also be corrupted if the memory used to hold saved processor state (usually located on the heap [5]) is freed and reallocated for other purposes.

Note that there are cases where the kernel makes explicit, legal, changes to the interrupted state of user-space code. For example, during signal handler dispatch, the kernel modifies interrupted program state that has been saved to memory, including the interrupted program's program counter and stack pointer [5]. Also, returning from a signal handler requires undoing the modifications made by signal delivery. The verifier must be able to distinguish legal from illegal changes to saved state.

2.2 Corrupting Stack State

The kernel directly manages the stacks of both user and kernel threads; it allocates and deallocates memory to hold them, sets up initial stack frames for new threads and signal handlers, and switches between stacks during a context switch or interrupt/system call return.

Memory for the stack is obtained from some standard memory allocation. Several safety violations are possible through this allocated memory. First, the memory for the

stack should only be used for stack frames created during normal function calls and not directly modified via arbitrary stores;³ such stores could corrupt the stack frames and thus compromise safety. Second, the memory for the stack must not be deallocated and reused for other memory objects while the stack is still in use. Third, a context switch must switch to a stack and its corresponding saved processor state as a pair; a context switch should not load processor state with the wrong stack or with a stack that has been deallocated. Fourth, after a stack is deallocated, live pointers to local variables allocated on the stack must not be dereferenced (the exiting thread may have stored pointers to such objects into global variables or the heap where they are accessible by other threads).

2.3 Corrupting Memory-Mapped I/O

Most systems today use memory-mapped I/O for controlling I/O devices and either memory-mapped I/O or DMA for performing data transfers. Many hardware architectures treat regular memory and memory-mapped I/O device memory (hereafter called I/O memory) identically, allowing a single set of hardware instructions to access both. From a memory safety perspective, however, it is better to treat regular memory and I/O memory as disjoint types of memory that are accessed using distinct instructions. First, I/O memory is not semantically the same as regular memory in that a load may not return the value last stored into the location; program analysis algorithms (used to enforce and optimize memory safety [15]) are not sound when applied to such memory. Second, I/O memory creates side-effects that regular memory does not. While erroneously accessing I/O memory instead of regular memory may not be a memory safety violation per se, it is still an error with potentially dire consequences. For example, the e1000e bug [9] caused fatal damage to hardware when an instruction (`cmpxchg`) that was meant to write to memory erroneously accessed memory-mapped I/O registers, which has undefined behavior. Therefore, for soundness of regular memory safety and for protection against a serious class of programming errors, it is best to treat regular memory and I/O memory as disjoint.

2.4 Corrupting Code

Besides the general memory corruption violations described below, there are only two ways in which the contents of code pages can be (or appear to be) corrupted. One is through self-modifying code (SMC); the other is through incorrect program loading operations (for new code or loadable kernel modules).

³An exception is when Linux stores the process's task structure at the bottom of the stack.

Self-modifying code directly modifies the sequence of instructions executed by the program. This can modify program behavior in ways not predicted by the compiler and hence bypass any of its safety checking techniques. For these reasons, most type-safe languages prohibit self-modifying code (which is distinct from “self-extending” behaviors like dynamic class loading). However, modern kernels use limited forms of self-modifying code for operations like enabling and disabling instrumentation [9] or optimizing synchronization for a specific machine configuration [8]. To allow such optimizations, the verifier must define limited forms of self-modifying code that do not violate the assumptions of the safety checker.

Second, the verifier must ensure that any program loading operation is implemented correctly. For example, any such operation, including new code, self-modifying code, or self-extending code (e.g., loadable kernel modules) requires flushing the instruction cache. Otherwise, cached copies of the old instructions may be executed out of the I-cache (and processors with split instruction/data caches may even execute old instructions with fresh data). This may lead to arbitrary memory safety violations for the kernel or application code.

2.5 General Memory Corruption

Finally, there are three kinds of kernel functionality that can corrupt arbitrary memory pages: (1) MMU configuration; (2) page swapping; and (3) DMA. Note that errors in any of these actions are generally invisible to a safety checking compiler and can violate the assumptions made by the compiler, as follows.

First, the kernel can violate memory safety with direct operations on virtual memory. Fundamentally, most of these are caused by creating an incorrect virtual-to-physical page mapping. Such errors include modifying mappings in the range of kernel stack memory, mapping the same physical page into two virtual pages (unintentionally), and changing a virtual-to-physical mapping for a live virtual page. As before, any of these errors can occur even with a type-safe language.

A second source of errors is in page swapping. When a page of data is swapped in on a page fault, memory safety can be violated if the data swapped in is not identical to the data swapped out from that virtual page. For example, swapping in the wrong data can cause invalid data to appear in pointers that are stored in memory.

Finally, a third source of problems is DMA. DMA introduces two problems. First, a DMA configuration error, device driver error, or device firmware error can cause a DMA transfer to overwrite arbitrary physical memory, violating type-safety assumptions. Second, even a correct DMA transfer may bring in unknown data which cannot be used in a type-safe manner, unless spe-

cial language support is added to enable that, e.g., to prevent such data being used as pointer values, as in the SPIN system [21].

3 Design Principles

We now describe the general design principles that a memory safe system can use to prevent the memory errors described in Section 2. As described earlier, we assume a safety checker already exists that creates a safe execution environment; the *verifier* is the set of extensions to the safety checker that enforces the underlying assumptions of the checker. Examples of safety checkers that could benefit directly from such extensions include SVA, SafeDrive, and XFI. We also assume that the kernel source code is available for modification.

Processor State: Preventing the corruption of processor state involves solving several issues. First, the verifier must ensure that the kernel does not make arbitrary changes to CPU registers. Most memory safe systems already do this by not providing instructions for such low-level modifications. Second, the verifier must ensure that processor state saved by a context switch, interrupt, trap, or system call is not accessed by memory load and store instructions. To do this, the verifier can allocate the memory used to store processor state within its own memory and allow the kernel to manipulate that state via special instructions that take an opaque handle (e.g., a unique integer) to identify which saved state buffer to use. For checkers like SVA and SafeDrive, the safety checker itself prevents the kernel from manufacturing and using pointers to these saved state buffers (e.g., via checks on accesses that use pointers cast from integers). Additionally, the verifier should ensure that the interface for context switching leaves the system in a known state, meaning that a context switch should either succeed completely or fail.

There are operations in which interrupted program state needs to be modified by the kernel (e.g., signal handler dispatch). The verifier must provide instructions for doing controlled modifications of interrupted program state; for example, it can provide an instruction to push function call frames on to an interrupted program’s stack [11]. Such instructions must ensure that either their modifications cannot break memory safety or that they only modify the saved state of interrupted user-space programs (modifying user-space state cannot violate the kernel’s memory safety).

Stack State: The memory for a kernel stack and for the processor state object (the in-memory representation of processor state) must be created in a single operation (instead of by separate operations), and the verifier should ensure that the kernel stack and processor state object

are always used and deallocated together. To ease implementation, it may be desirable to move some low-level, error-prone stack and processor state object initialization code into the verifier. The verifier must also ensure that memory loads and stores do not modify the kernel stack (aside from accessing local variables) and that local variables stored on the stack can no longer be accessed when the kernel stack is destroyed.

Memory-mapped I/O: The verifier must require that all I/O object allocations be identifiable in the kernel code, (e.g., declared via a pseudo-allocator). It should also ensure that only special I/O read and write instructions can access I/O memory (these special instructions can still be translated into regular memory loads and stores for memory-mapped I/O machines) and that these special instructions cannot read or write regular memory objects. If the verifier uses type-safety analysis to optimize run-time checks, it should consider I/O objects (objects analogous to memory objects but that reside in memory-mapped I/O pages) to be *type-unsafe* as the device's firmware may use the I/O memory in a type-unsafe fashion. Since it is possible for a pointer to point to both I/O objects and memory objects, the verifier should place run-time checks on such pointers to ensure that they are accessing the correct type of object (memory or I/O), depending upon the operation in which the pointer is used.

Kernel Code: The verifier must not permit the kernel to modify its code segment. However, it can support a limited version of self-modifying code that is easy to implement and able to support the uses of self-modifying code found in commodity kernels. In our design, the kernel can specify regions of code that can be enabled and disabled. The verifier will be responsible for replacing native code with no-op instructions when the kernel requests that code be disabled and replacing the no-ops with the original code when the kernel requests the code to be re-enabled. When analyzing code that can be enabled and disabled, the verifier can use conservative analysis techniques to generate results that are correct regardless of whether the code is enabled or disabled. For example, our pointer analysis algorithm, like most other inter-procedural ones used in production compilers, computes a *may-points-to* result [24], which can be computed with the code enabled; it will still be correct, though perhaps conservative, if the code is disabled.

To ensure that the instruction cache is properly flushed, our design calls for the safety checker to handle all translation to native code. The safety checker already does this in JVMs, safe programming languages, and in the SVA system [10]. By performing all translation to native code, the verifier can ensure that all appropriate CPU caches are flushed when new code is loaded into the system.

General Memory Corruption: The verifier must implement several types of protection to handle the general memory corruption errors in Section 2.5.

MMU configuration: To prevent MMU misconfiguration errors, the verifier must be able to control access to hardware page tables or processor TLBs and vet changes to the MMU configuration before they are applied. Implementations can use para-virtualization techniques [16] to control the MMU. The verifier must prevent pages containing kernel memory objects from being made accessible to non-privileged code and ensure that pages containing kernel stack frames are not mapped to multiple virtual addresses (i.e., double mapped) or unmapped before the kernel stack is destroyed.⁴ Verifiers optimizing memory access checks must also prohibit double mappings of pages containing *type known* objects; this will prevent data from being written into the page in a way that is not detected by compiler analysis techniques. Pages containing *type-unknown* memory objects can be mapped multiple times since run-time checks already ensure that the data within them does not violate any memory safety properties. The verifier must also ensure that MMU mappings do not violate any other analysis results upon which optimizations depend.

Page swapping: For page swapping, the kernel must notify the verifier before swapping a page out (if not, the verifier will detect the omission on a subsequent physical page remapping operation). The verifier can then record any metadata for the page as well as a checksum of the contents and use these when the page is swapped back in to verify that the page contents have not changed.

DMA: The verifier should prevent DMA transfers from overwriting critical memory such as the kernel's code segment, the verifier's code and data, kernel stacks (aside from local variables), and processor state objects. Implementation will require the use of IOMMU techniques like those in previous work [17, 36]. Additionally, if the verifier uses type information to optimize memory safety checks, it must consider the memory accessible via DMA as *type-unsafe*. This solution is strictly stronger than previous work (like that in SPIN [21]): it allows pointer values in input data whereas they do not (and they do not guarantee type safety for other input data).

Entry Points: To ensure control-flow integrity, the kernel should not be entered in the middle of a function. Therefore, the verifier must ensure that all interrupt, trap, and system call handlers registered by the kernel are the initial address of a valid function capable of servicing the interrupt, trap, or system call, respectively.

⁴We assume the kernel does not swap stack pages to disk, but the design can be extended easily to allow this.

4 Background: Secure Virtual Architecture

The Secure Virtual Architecture (SVA) system (Figure 1) places a compiler-based virtual machine between the processor and the traditional software stack [10, 11]. The virtual machine (VM) presents a virtual instruction set to the software stack and translates virtual instructions to the processor's native instruction set either statically (the default) or dynamically. The virtual instruction set is based on the LLVM code representation [23], which is designed to be low-level and language-independent, but still enables sophisticated compiler analysis and transformation techniques. This instruction set can be used for both user-space and kernel code [11].

SVA optionally provides strong safety guarantees for C/C++ programs compiled to its virtual instruction set, close to that of a safe language. The key guarantees are:

1. *Partial type safety*: Operations on a subset of data are type safe.
2. *Memory safety*: Loads and stores only access the object to which the dereferenced pointer initially pointed, and within the bounds of that object.
3. *Control flow integrity*: The kernel code only follows execution paths predicted by the compiler; this applies to both branches and function calls.
4. *Tolerating dangling pointers*: SVA does not detect uses of dangling pointers but *guarantees that they are harmless*, either via static analysis (for type-safe data) or by detecting violations through run-time checks (for non-type safe data).
5. *Sound operational semantics*: SVA defines a virtual instruction set with an operational semantics that is guaranteed not to be violated by the kernel code; sound program analysis or verification tools can be built on this semantics.

Briefly, SVA provides these safety guarantees as follows. First, it uses a pointer analysis called Data Structure Analysis (DSA) [24] to partition memory into logical partitions (“points to sets”) and to check which partitions are always accessed or indexed with a single type. These partitions are called “type-known” (TK); the rest are “type-unknown” (TU). SVA then creates a run-time representation called a “metapool” for each partition. It maintains a lookup table in each metapool of memory objects and their bounds to support various run-time checks. Maintaining a table per metapool instead of a single global table greatly improves the performance of the run-time checks [14].

Compile-time analysis with DSA guarantees that all TK partitions are type-safe. Moreover, all uses of data

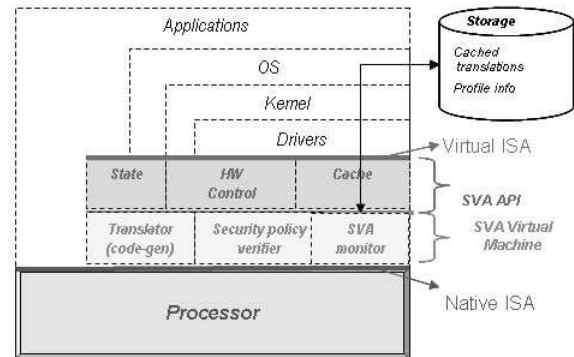


Figure 1: System Organization with SVA [10]

and function pointers loaded *out of* TK partitions are type safe. SVA simply has to ensure that dangling pointer references to TK metapools cannot create a type violation by enforcing two constraints: (a) objects in TK metapools are aligned identically; and (b) freed memory from such a metapool is never used for a different metapool until the former is destroyed. These constraints are enforced by modifying the kernel allocators manually during the process of porting the kernel to SVA; this means that the allocators are effectively trusted and not checked. To enforce these constraints for stack objects belonging to TK metapools, SVA automatically modifies the kernel code to allocate such objects on the heap. Together, these guarantee that a pointer to a freed object and a new object (including array elements) access values of identical type [15].

At run-time, the SVA VM (thereafter called VM) performs a number of additional checks and operations. All globals and allocated objects are registered in the metapool to which they belong (derived from the target partition of the return pointer). Loads and stores that use pointers loaded from TU metapools are checked by looking up the target address in the metapool lookup table. Note that this works whether or not the pointer value is a dangling pointer, and even for pointers “manufactured” by casting arbitrary integers. Similarly, it checks function pointers obtained from TU metapools to ensure that they only access one of the target functions of that pointer predicted by DSA. Run-time checks also ensure that pointers to TK objects that are loaded from TU memory objects are checked since a TU object may have an invalid value for the TK pointer. All array indexing operations for TK or TU metapools are checked in the lookup table, which records the bounds for each object [14]⁵.

Note that the VM relies on the safe execution environ-

⁵Note that we permit a pointer to “leave” its target object and later return, as long as it is not accessed while it is out of bounds [32].

ment to protect the VM code and data memory instead of using the MMU and incurring the cost of switching page tables on every VM invocation. Since the environment prevents access to unregistered data objects or outside the bounds of legal objects, we can simply monitor all run-time kernel object registrations and ensure that they do not reside in VM code or data pages.

A subset of the SVA instruction set, SVA-OS, provides instructions designed to support an operating system's special interaction with the hardware [10, 11]. These include instructions for loading from/storing to I/O memory, configuring the MMU, and manipulating program state. An important property is that a kernel ported to SVA using the SVA-OS instructions *contains no assembly code*; this simplifies the compiler's task of safety checking within SVA. Nevertheless, these instructions provide low-level hardware interactions that can generate all the problems described in Section 2 if used incorrectly; it is very difficult for the compiler to check their correct use in the original design. In particular, the VM does not perform any special checks for processor state objects, direct stack manipulation, memory mapped I/O locations, MMU configuration changes, or DMA operations. Also, it disallows self-modifying code.

For example, we tested two [39, 42] of the three reported low-level errors we found for Linux 2.4.22, the kernel version ported to SVA (we could not try the third [3] for reasons explained in Section 7.1). Although both are memory safety violations, *neither of them was detected or prevented by the original SVA*.

5 Design

Our design is an extension of the original Secure Virtual Architecture (SVA) described in Section 4. SVA provides strong memory safety guarantees for kernel code and an abstraction of the hardware that is both low-level (e.g., context switching, I/O, and MMU configuration policies are still implemented in the kernel), yet easy to analyze (because the SVA-OS instructions for interacting with hardware are slightly higher level than typical processor instructions). Below, we describe our extensions to provide memory safety in the face of errors in kernel-hardware interactions.

5.1 Context Switching

Previously, the SVA system performed context switching using the `sva_load_integer` and `sva_save_integer` instructions [10], which saved from and loaded into the processor the processor state (named Integer State). These instructions stored processor state in a kernel allocated memory buffer which could be later modified by memory-safe store instructions or freed by

the kernel deallocator. Our new design calls a single instruction named `sva_swap_integer` (see Table 1) that saves the old processor state and loads the new state in a single operation.

This design has all of the necessary features to preserve memory safety when context switching. The `sva_swap_integer` instruction allocates the memory buffer to hold processor state within the VM's memory and returns an opaque integer identifier which can be used to re-load the state in a subsequent call to `sva_swap_integer`. Combined with SVA's original protections against manufactured pointers, this prevents the kernel from modifying or deallocating the saved processor state buffer. The design also ensures correct deallocation of the memory buffer used to hold processor state. The VM tracks which identifiers are mapped to allocated state buffers created by `sva_swap_integer`; these memory buffer/identifier pairs are kept alive until the state is placed back on the processor by another call to `sva_swap_integer`. Once state is placed back on the processor, the memory buffer is deallocated, and the identifier invalidated to prevent the kernel from trying to restore state from a deallocated state buffer.

Finally, `sva_swap_integer` will either succeed to context switch and return an identifier for the saved processor state, or it will fail, save no processor state, and continue execution of the currently running thread. This ensures that the kernel stack and the saved processor state are always synchronized.

5.2 Thread Management

A thread of execution consists of a stack and a saved processor state that can be used to either initiate or continue execution of the thread. Thread creation is therefore comprised of three operations: allocating memory for the new thread's stack, initializing the new stack, and creating an initial state that can be loaded on to the processor using `sva_swap_integer`.

The VM needs to know where kernel stacks are located in order to prevent them from being written by load and store instructions. We introduce a new SVA instruction, `sva_declare_stack`, which a kernel uses to declare that a memory object will be used as a stack. During pointer analysis, any pointers passed to `sva_declare_stack` and pointers that alias with such pointers are marked with a special *DeclaredStack* flag; this flag indicates that run-time checks are needed on stores via such pointers to ensure that they are not writing into a kernel stack. The compiler, on seeing an `sva_declare_stack` instruction, will also verify, statically (via pointer analysis) if possible but at run-time if necessary, that the memory object used for the new stack is either a global or heap object; this will prevent

Name	Description
<code>sva_swap_integer</code>	Saves the current processor state into an internal memory buffer, loads previously saved state referenced by its ID, and returns the ID of the new saved state.
<code>sva_declare_stack</code>	Declares that a memory object is to be used as a new stack.
<code>sva_release_stack</code>	Declares that a memory object is no longer used as a stack.
<code>sva_init_stack</code>	Initializes a new stack.

Table 1: SVA Instructions for Context Switching and Thread Creation.

stacks from being embedded within other stacks. After this check is done, `sva_declare_stack` will unregis-
ter the memory object from the set of valid memory ob-
jects that can be accessed via loads and stores and record
the stack’s size and location within the VM’s internal
data structures as a valid kernel stack.

To initialize a stack and the initial processor state
that will use the memory as a stack, we introduce
`sva_init_stack`; this instruction will initialize the
stack and create a new saved Integer State which can
be used in `sva_swap_integer` to start executing
the new thread. The `sva_init_stack` instruction
verifies (either statically or at run-time) that its argu-
ment has previously been declared as a stack using
`sva_declare_stack`. When the new thread wakes
up, it will find itself running within the function specified
by the call to `sva_init_stack`; when this function re-
turns, it will return to user-space at the same location as
the original thread entered.

Deleting a thread is composed of two operations. First,
the memory object containing the stack must be deal-
located. Second, any Integer State associated with the
stack that was saved on a context switch must be in-
validated. When the kernel wishes to destroy a thread,
it must call the `sva_release_stack` instruction; this
will mark the stack memory as a regular memory object
so that it can be freed and invalidates any saved Integer
State associated with the stack.

When a kernel stack is deallocated, there may be
pointers in global or heap objects that point to mem-
ory (i.e., local variables) allocated on that stack. SVA
must ensure that dereferencing such pointers does not
violate memory safety. Type-unsafe stack allocated ob-
jects are subject to load/store checks and are registered
with the SVA virtual machine [10]. In order for the
`sva_release_stack` instruction to invalidate such
objects when stack memory is reclaimed, the VM records
information on stack object allocations and associates
this information with the metadata about the stack in
which the object is allocated. In this way, when a stack is
deallocated, any live objects still registered with the vir-
tual machine are automatically invalidated as well; run-
time checks will no longer consider these stack allocated
objects to be valid objects. Type-known stack allocated
objects can never be pointed to by global or heap objects;
SVA already transforms such stack allocations into heap

allocations [15, 10] to make dangling pointer dereferenc-
ing to type-known stack allocated objects safe [15].

5.3 Memory Mapped I/O

To ensure safe use of I/O memory, our system must be
able to identify where I/O memory is located and when
the kernel is legitimately accessing it.

Identifying the location of I/O memory is straightfor-
ward. In most systems, I/O memory is located at (or
mapped into) known, constant locations within the sys-
tem’s address space, similar to global variables. In some
systems, a memory-allocator-like function may remap
physical page frames corresponding to I/O memory to
a virtual memory address [5]. The insight is that I/O
memory is grouped into objects just like regular mem-
ory; in some systems, such I/O objects are even allocated
and freed like heap objects (e.g., Linux’s `ioremap()`
function [5]). To let the VM know where I/O memory
is located, we must modify the kernel to use a pseudo-
allocator that informs the VM of global I/O objects; we
can also modify the VM to recognize I/O “allocators”
like `ioremap()` just like it recognizes heap allocators
like Linux’s `kmalloc()` [5].

Given this information, the VM needs to determine
which pointers may point to I/O memory. To do so,
we modified the SVA points-to analysis algorithm [24]
to mark the target (i.e., the “points-to set”) of a pointer
holding the return address of the I/O allocator with a spe-
cial *I/O flag*. This also flags other pointers aliased to
such a pointer because any two aliased pointers point to
a common target [24].

We also modified the points-to analysis to mark I/O
memory as *type-unknown*. Even if the kernel accesses
I/O memory in a type-consistent fashion, the firmware
on the I/O device may not. *Type-unknown* memory in-
curs additional run-time checks but allows kernel code
to safely use pointer values in such memory as pointers.

We also extended SVA to record the size and virtual
address location of every I/O object allocation and deal-
location by instrumenting every call to the I/O allocator
and deallocator functions. At run-time, the VM records
these I/O objects in a per-metapool data structure that
is disjoint from the structure used to record the bounds
of regular memory objects. The VM also uses new run-
time checks for checking I/O load and store instructions.

Since I/O pointers can be indexed like memory pointers (an I/O device may have an array of control registers), the bounds checking code must check both regular memory objects and I/O memory objects. Load and store checks on regular memory pointers *without the I/O flag* remain unchanged; they only consider memory objects. New run-time checks are needed on both memory and I/O loads and stores for pointers that have both the I/O flag and one or more of the memory flags (heap, stack, global) to ensure that they only access regular or I/O memory objects, respectively.

5.4 Safe DMA

We assume the use of an IOMMU for preventing DMA operations from overflowing object bounds or writing to the wrong memory address altogether [13]. The SVA virtual machine simply has to ensure that the I/O MMU is configured so that DMA operations cannot write to the virtual machine's internal memory, kernel code pages, pages which contain type-safe objects, and stack objects.

We mark all memory objects that may be used for DMA operations as type-unsafe, similar to I/O memory that is accessed directly. We assume that any pointer that is *stored into* I/O memory is a potential memory buffer for DMA operations. We require alias analysis to identify such stores; it simply has to check that the target address is in I/O memory and the store value is of pointer type. We then mark the points-to set of the store value pointer as *type-unknown*.

5.5 Virtual Memory

Our system must control the MMU and vet changes to its configuration to prevent safety violations and preserve compiler-inferred analysis results. Below, we describe the mechanism by which our system monitors and controls MMU configuration and then discuss how we use this mechanism to enforce several safety properties.

5.5.1 Controlling MMU Configuration

SVA provides different MMU interfaces for hardware TLB processors and software TLB processors [11]. For brevity, we describe only the hardware TLB interface and how our design uses it to control MMU configuration.

The SVA interface for hardware TLB systems (given in Table 2) is similar to those used in VMMs like Xen [16] and is based off the `paravirtops` interface [50] found in Linux 2.6. The page table is a 3-level page table, and there are instructions for changing mappings at each level. In this design, the OS first tells the VM which memory pages will be used for the page table (it must specify at what level the page will

appear in the table); the VM then takes control of these pages by zeroing them (to prevent stale mappings from being used) and marking them read-only to prevent the OS from accessing them directly. The OS must then use special SVA instructions to update the translations stored in these page table pages; these instructions allow SVA to first inspect and modify translations before accepting them and placing them into the page table. The `sva_load_pagetable` instruction selects which page table is in active use and ensures that only page tables controlled by SVA are ever used by the processor. This interface, combined with SVA's control-flow integrity guarantees [10], ensure that SVA maintains control of all page mappings on the system.

5.5.2 Memory Safe MMU Configuration

For preventing memory safety violations involving the MMU, the VM needs to track two pieces of information. First, the VM must know the purpose of various ranges of the virtual address space; the kernel must provide the virtual address ranges of user-space memory, kernel data memory, and I/O object memory. This information will be used to prevent physical pages from being mapped into the wrong virtual addresses (e.g., a memory mapped I/O device being mapped into a virtual address used by a kernel memory object). A special instruction permits the kernel to communicate this information to the VM.

Second, the VM must know how physical pages are used, how many times they are mapped into the virtual address space, and whether any MMU mapping makes them accessible to unprivileged (i.e., user-space) code. To track this information, the VM associates with each physical page a set of flags and counters. The first set of flags are mutually exclusive and indicate the purpose of the page; a page can be marked as: L1 (Level-1 page table page), L2 (Level-2 page table page), L3 (Level-3 page table page), RW (a standard kernel page holding memory objects), IO (a memory mapped I/O page), `stack` (kernel stack), `code` (kernel or SVA code), or `svamem` (SVA data memory). A second flag, the TK flag, specifies whether a physical page contains *type-known* data. The VM also keeps a count of the number of virtual pages mapped to the physical page and a count of the number of mappings that make the page accessible to user-space code.

The flags are checked and updated by the VM whenever the kernel requests a change to the page tables or performs relevant memory or I/O object allocation. Calls to the memory allocator are instrumented to set the RW and, if appropriate, the TK flag on pages backing the newly allocated memory object. On system boot, the VM sets the IO flag on physical pages known to be memory-mapped I/O locations. The `stack`

Name	Description
<code>sva_end_mem_init</code>	End of the virtual memory boot initialization. Flags all page table pages, and mark them read-only.
<code>sva_declare_l1_page</code>	Zeroes the page and flags it read-only and L1.
<code>sva_declare_l2_page</code>	Zeroes the page and flags it read-only and L2.
<code>sva_declare_l3_page</code>	Puts the default mappings in the page and flags it read-only and L3.
<code>sva_remove_l1_page</code>	Unflags the page read-only and L1.
<code>sva_remove_l2_page</code>	Unflags the page read-only and L2.
<code>sva_remove_l3_page</code>	Unflags the page read-only and L3.
<code>sva_update_l1_mapping</code>	Updates the mapping if the mapping belongs to an L1 page and the page is not already mapped for a type known pool, sva page, code page, or stack page.
<code>sva_update_l2_mapping</code>	Updates the mapping if the mapping belongs to an L2 page and the new mapping is for an L1 page.
<code>sva_update_l3_mapping</code>	Updates the mapping if the mapping belongs to an L3 page and the new mapping is for an L2 page.
<code>sva_load_pagetable</code>	Check that the physical page is an L3 page and loads it in the page table register.

Table 2: MMU Interface for a Hardware TLB Processor.

flag is set and cleared by `sva_declare_stack` and `sva_release_stack`, respectively. Changes to the page table via the instructions in Table 2 update the counters and the L1, L2, and L3 flags.

The VM uses all of the above information to detect, at run-time, violations of the safety requirements in Section 3. Before inserting a new page mapping, the VM can detect whether the new mapping will create multiple mappings to physical memory containing *type-known* objects, map a page into the virtual address space of the VM or kernel code segment, unmap or double map a page that is part of a kernel stack, make a physical page containing kernel memory accessible to user-space code, or map memory-mapped I/O pages into a kernel memory object (or vice-versa). Note that SVA currently trusts the kernel memory allocators to (i) return different virtual addresses for every allocation, and (ii) not to move virtual pages from one metapool to another until the original metapool is destroyed.

5.6 Self-modifying Code

The new SVA system supports the restricted version of self-modifying code described in Section 3: OS kernels can disable and re-enable pre-declared pieces of code. SVA will use compile-time analysis carefully to ensure that replacing the code with no-op instructions will not invalidate the analysis results.

We define four new instructions to support self-modifying code. The first two instructions, `sva_begin_alt` and `sva_end_alt` enclose the code regions that may be modified at runtime. They must be properly nested and must be given a unique identifier. The instructions are not emitted in the native code. The two other instructions, `sva_disable_code` and `sva_enable_code` execute at runtime. They take the identifier given to the `sva_begin_alt` and `sva_end_alt` instructions. `sva_disable_code` saves the previous code and inserts no-ops in the code, and `sva_enable_code` restores the previous code.

With this approach, SVA can support most uses of self-modifying code in operating systems. For instance, it supports the *alternatives*⁶ framework in Linux 2.6 [8] and Linux’s *ftrace* tracing support [9] which disables calls to logging functions at run-time.

5.7 Interrupted State

On an interrupt, trap, or system call, the original SVA system saves processor state within the VM’s internal memory and permits the kernel to use specialized instructions to modify the state via an opaque handle called the interrupt context [10, 11]. These instructions, which are slightly higher-level than assembly code, are used by the kernel to implement operations like signal handler dispatch and starting execution of user programs. Since systems such as Linux can be interrupted while running kernel code [5], these instructions can violate the kernel’s memory safety if used incorrectly on interrupted kernel state. To address these issues, we introduce several changes to the original SVA design.

First, we noticed that all of the instructions that manipulate interrupted program state are either memory safe (e.g., the instruction that unwinds stack frames for kernel exception handling [11]) or only need to modify the interrupted state of user-space programs. Hence, all instructions that are not intrinsically memory safe will verify that they are modifying interrupted user-space program state. Second, the opaque handle to the interrupt context will be made implicit so that no run-time checks are needed to validate it when it is used. We have observed that the Linux kernel only operates upon the most recently created interrupt context; we do not see a need for other operating systems of similar design to do so, either. Without an explicit handle to the interrupt context’s location in memory, no validation code is needed, and the kernel cannot create a pointer to the saved program state (except for explicit integer to pointer casts, uses of which will be caught by SVA’s existing checks) [10].

⁶Linux 2.6, file `include/asm-x86/alternative.h`

5.8 Miscellaneous

To ensure control-flow integrity requirements, the VM assumes control of the hardware interrupt descriptor table; the OS kernel must use special instructions to associate a function with a particular interrupt, trap, or system call [11, 29]. Similar to indirect function call checks, SVA can use static analysis and run-time checks to ensure that only valid functions are registered as interrupt, trap, or system call handlers.

SVA provides two sets of atomic memory instructions: `sva_fetch_and_phi` where `phi` is one of several integer operations (e.g., `add`), and `sva_compare_and_swap` which performs an atomic compare and swap. The static and run-time checks that protect regular memory loads and stores also protect these operations.

6 Modifications to the Linux Kernel

We implemented our design by improving and extending the original SVA prototype and the SVA port of the Linux 2.4.22 kernel [10]. The previous section described how we modified the SVA-OS instructions. Below, we describe how we modified the Linux kernel to use these new instructions accordingly. We modified less than 100 lines from the original SVA kernel to port our kernel to the new SVA-OS API; the original port of the i386 Linux kernel to SVA modified 300 lines of architecture-independent code and 4,800 lines of architecture-dependent code [10].

6.1 Changes to Baseline SVA

The baseline SVA system in our evaluation (Section 7) is an improved version of the original SVA system [10] that is suitable for determining the extra overhead incurred by the run-time checks necessitated by the design in Section 5. First, we fixed several bugs in the optimization of run-time checks. Second, while the original SVA system does not analyze and protect the whole kernel, there is no fundamental reason why it cannot. Therefore, we chose to disable optimizations which apply only to incomplete kernel code for the experiments in Section 7. Third, the new baseline SVA system recognizes `ioremap()` as an allocator function even though it does not add run-time checks for I/O loads and stores. Fourth, we replaced most uses of the `_get_free_pages()` page allocator with `kmalloc()` in code which uses the page allocator like a standard memory allocator; this ensures that most kernel allocations are performed in kernel pools (i.e., `kmem_cache_ts`) which fulfill the requirements for allocators as described in the original SVA work [10].

We also modified the SVA Linux kernel to use the new SVA-OS instruction set as described below. This ensured

that the only difference between our baseline SVA system and our SVA system with the low-level safety protections was the addition of the run-time checks necessary to ensure safety for context switching, thread management, MMU, and I/O memory safety.

6.2 Context Switching/Thread Creation

The modifications needed for context switching were straightforward. We simply modified the `switch_to` macro in Linux [5] to use the `sva_swap_integer` instruction to perform context switching.

Some minor kernel modifications were needed to use the new thread creation instructions. The original i386 Linux kernel allocates a single memory object which holds both a thread's task structure and the kernel stack for the thread [5], but this cannot be done on our system because `sva_declare_stack` requires that a stack consumes an entire memory object. For our prototype, we simply modified the Linux kernel to perform separate allocations for the kernel stack and the task structure.

6.3 I/O

As noted earlier, our implementation enhances the pointer analysis algorithm in SVA (DSA [24]) to mark pointers that may point to I/O objects. It does this by finding calls to the Linux `_ioremap()` function. To make implementation easier, we modified `ioremap()` and `ioremap_nocache()` in the Linux source to be macros that call `_ioremap()`.

Our test system's devices do not use global I/O memory objects, so we did not implement a pseudo allocator for identifying them. Also, we did not modify DSA to mark memory stored into I/O device memory as type-unknown. The difficulty is that Linux casts pointers into integers before writing them into I/O device memory. The DSA implementation does not have solid support for tracking pointers through integers i.e., it does not consider the case where an integer may, in fact, be pointing to a memory object. Implementing these changes to provide DMA protection is left as future work.

6.4 Virtual Memory

We implemented the new MMU instructions and run-time checks described in Section 5.5 and ported the SVA Linux kernel to use the new instructions. Linux already contains macros to allocate, modify and free page table pages. We modified these macros to use our new API (which is based on the `paravirtops` interface from Linux 2.6). We implemented all of the run-time checks except for those that ensure that I/O device memory isn't

mapped into kernel memory objects. These checks require that the kernel allocate all I/O memory objects within a predefined range of the virtual address space, which our Linux kernel does not currently do.

7 Evaluation and Analysis

Our evaluation has two goals. First, we wanted to determine whether our design for low-level software/hardware interaction was effective at stopping security vulnerabilities in commodity OS kernels. Second, we wanted to determine how much overhead our design would add to an already existing memory-safety system.

7.1 Exploit Detection

We performed three experiments to verify that our system catches low-level hardware/software errors: First, we tried two different exploits on our system that were reported on Linux 2.4.22, the Linux version that is ported to SVA. The exploits occur in the MMU subsystem; both give an attacker root privileges. Second, we studied the e1000e bug [9]. We could not duplicate the bug because it occurs in Linux 2.6, but we explain why our design would have caught the bug if Linux 2.6 had been ported to SVA. Third, we inserted many low-level operation errors inside the kernel to evaluate whether our design prevents the safety violations identified in Section 2.

Linux 2.4.22 exploits. We have identified three reported errors for Linux 2.4.22 caused by low-level kernel-hardware interactions [3, 39, 42]. Our experiment is limited to these errors because we needed hardware/software interaction bugs that were in Linux 2.4.22. Of these, we could not reproduce one bug due to a lack of information in the bug report [3]. The other two errors occur in the `mremap` system call but are distinct errors.

The first exploit [42] is due to an overflow in a count of the number of times a page is mapped. The exploit code overflows the counter by calling `fork`, `mmap`, and `mremap` a large number of times. It then releases the page, giving it back to the kernel. However, the exploit code still has a reference to the page; therefore, if the page is reallocated for kernel use, the exploit code can read and modify kernel data. Our system catches this error because it disallows allocating kernel objects in a physical page mapped in user space.

The second exploit [39] occurs because of a missing error check in `mremap` which causes the kernel to place page table pages with valid page table entries into the page table cache. However, the kernel assumes that page table pages in the page table cache do not contain any entries. The exploit uses this vulnerability by calling `mmap`, `mremap` and `munmap` to release a page table

page with page entries that contain executable memory. Then, on an `exec` system call, the linker, which executes with root privileges, allocates a page table page, which happens to be the previously released page. The end result is that the linker jumps to the exploit's executable memory and executes the exploit code with root privileges. The SVA VM prevents this exploit by always zeroing page table pages when they are placed in a page directory so that no new, unintended, memory mappings are created for existing objects.

The e1000e bug. The fundamental cause of the e1000e bug is a memory load/store (the x86 `cmpxchg` instruction) on a dangling pointer, which happens to point to an I/O object. The `cmpxchg` instruction has non-deterministic behavior on I/O device memory and may corrupt the hardware. The instruction was executed by the `ftrace` subsystem, which uses self-modifying code to trace the kernel execution. It took many weeks for skilled engineers to track the problem. With our new safety checks, SVA would have detected the bug at its first occurrence. The self-modifying code interface of SVA-OS only allows enabling and disabling of code; writes to what the kernel (incorrectly) thought was its code is not possible. SVA actually has a second line of defense if (hypothetically) the self-modifying code interface did not detect it: SVA would have prevented the I/O memory from being mapped into code pages, and thus prevented this corruption. (And, hypothetically again, if a dangling pointer to a data object had caused the bug, SVA would have detected any ordinary reads and writes trying to write to I/O memory locations.)

Kernel error injection. To inject errors, we added new system calls into the kernel; each system call triggers a specific kind of kernel/hardware interaction error that either corrupts memory or alters control flow. We inserted four different errors. The first error modifies the saved Integer State of a process so that an invalid Integer State is loaded when the process is scheduled. The second error creates a new MMU mapping of a page containing type-known kernel memory objects and modifies the contents of the page. The third error modifies the MMU mappings of pages in the stack range. The fourth error modifies the internal metadata of SVA to set incorrect bounds for all objects. This last error shows that with the original design, we can *disable the SVA memory safety checks that prevent Linux exploits*; in fact, it would not be difficult to do so with this bug alone for three of the four kernel exploits otherwise prevented by SVA [10].

All of the injected errors were caught by the new SVA implementation. With the previous implementation, these errors either crash the kernel or create undefined behavior. This gives us confidence about the correctness of our new design and implementation of SVA. Note that

we only injected errors that our design addresses because we believe that our design is “complete” in terms of the possible errors due to kernel-hardware interactions. Nevertheless, the injection experiments are useful because they *validate that the design and implementation actually solve these problems*.

7.2 Performance

To determine the impact of the additional run-time checks on system performance, we ran several experiments with applications typically used on server and end-user systems. We ran tests on the original Linux 2.4.22 kernel (marked i386 in the figures and tables), the same kernel with the original SVA safety checks [10] (marked SVA), and the SVA kernel with our safety checks for low-level software/hardware interactions (marked SVA-OS).

It is important to note that an underlying memory safety system like SVA can incur significant run-time overhead for C code, especially for a commodity kernel like Linux that was not designed for enforcement of memory safety. Such a system is not the focus of this paper. Although we present our results relative to the original (unmodified) Linux/i386 system for clarity, we focus the discussion on the excess overheads introduced by SVA-OS beyond those of SVA since the new techniques in SVA-OS are the subject of the current work.

We ran these experiments on a dual-processor AMD Athlon 2100+ at 1,733 MHz with 1 GB of RAM and a 1 Gb/s network card. We configured the kernel as an SMP kernel but ran it in on a single processor since the SVA implementation is not yet SMP safe. Network experiments used a dedicated 1 Gb/s switch. We ran our experiments in single-user mode to prevent standard system services from adding noise to our performance numbers.

We used several benchmarks in our experiments: the `thttpd` Web server, the OpenSSH `sshd` encrypted file transfer service, and three local applications – `bzip2` for file compression, the lame MP3 encoder, and a perl interpreter. These programs have a range of different demands on kernel operations. Finally, to understand why some programs incur overhead while others do not, we used a set of microbenchmarks including the HBenchoS microbenchmark suite [6] and two new tests we wrote for the poll and select system calls.

Application Performance First, we used ApacheBench to measure the file-transfer bandwidth of the `thttpd` web server [31] serving static HTML pages. We configured ApacheBench to make 5000 requests using 25 simultaneous connections. Figure 2 shows the results of both the original SVA kernel and the SVA kernel with the new run-time checks described in Section 5. Each bar is the average bandwidth of 3 runs of the experiment; the results are normalized to the

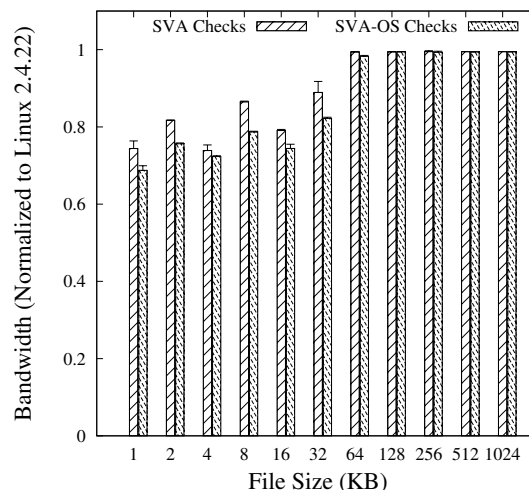


Figure 2: Web Server Bandwidth (Linux/i386 = 1.0)

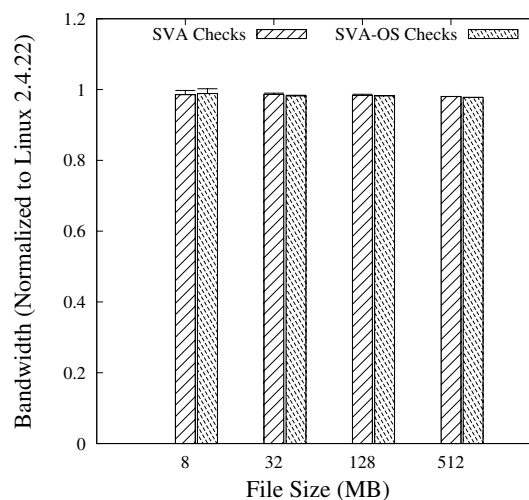


Figure 3: SSH Server Bandwidth (Linux/i386 = 1.0)

original i386 Linux kernel. For small files (1 KB - 32 KB) in which the original SVA system adds significant overhead, our new run-time checks incur a small amount of additional overhead (roughly a 9% decrease in bandwidth relative to the SVA kernel). However, for larger file sizes (64 KB or more), the SVA-OS checks add negligible overhead to the original SVA system.

We also measured the performance of `sshd`, a login server offering encrypted file transfer. For this test, we measured the bandwidth of transferring several large files from the server to our test client; the results are shown in Figure 3. For each file size, we first did a priming run to bring file system data into the kernel’s buffer cache; subsequently, we transferred the file three times. Figure 3 shows the mean of the receive bandwidth of the three runs normalized to the mean receive bandwidth mea-

Benchmark	i386 (s)	SVA (s)	SVA-OS (s)	% Increase from i386 to SVA-OS	Description
bzip2	18.7 (0.47)	18.3 (0.47)	18.0 (0.00)	0.0%	Compressing 64 MB file
lame	133.3 (3.3)	132 (0.82)	126.0 (0.82)	-0.1%	Converting 206 MB WAV file to MP3
perl	22.3 (0.47)	22.3 (0.47)	22.3 (0.47)	0.0%	Interpreting scrabbl.pl from SPEC 2000

Table 3: Latency of Applications. Standard Deviation Shown in Parentheses.

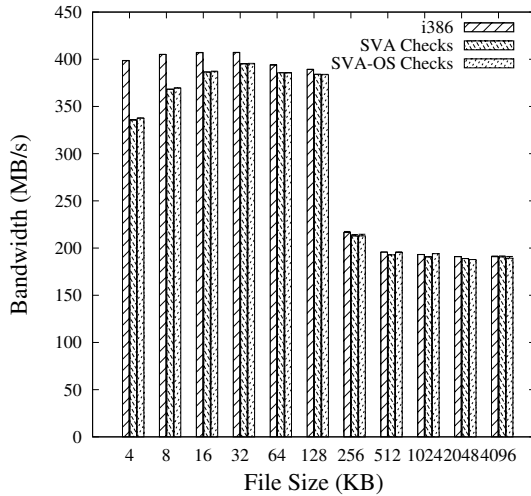


Figure 4: File System Bandwidth

Benchmark	i386 (s)	SVA (s)	SVA-OS (s)
bzip2	41	40	40
lame	203	202	202
perl	24	23	23

Table 4: Latency of Applications During Priming Run.

sured on the original i386 kernel; note that the units on the X-axis are MB. Our results indicate that there is no significant decrease in bandwidth due to the extra run-time checks added by the original SVA system or the new run-time checks presented in this paper. This outcome is far better than `thttpd`, most likely due to the large file sizes we transferred via `scp`. For large file sizes, the network becomes the bottleneck: transferring an 8 MB file takes 62.5 ms on a Gigabit network, but the overheads for basic system calls (shown in Table 5) show overheads of only tens of microseconds.

To see what effect our system would have on end-user application performance, we ran experiments on the client-side programs listed in Table 3. We tested `bzip2` compressing a 64 MB file, the LAME MP3 encoder converting a 206 MB file from WAV to MP3 format, and the `perl` interpreter running the training input from the SPEC 2000 benchmark suite. For each test, we ran the program once to prime any caches within the operating system and then ran each program three times. Table 3 shows the average of the execution times of the three runs and the

percent overhead that the applications experienced executing on the SVA-OS kernel relative to the original i386 Linux kernel. The results show that our system adds virtually no overhead for these applications, even though some of the programs (`bzip2` and `lame`) perform substantial amounts of I/O. Table 4 shows the latency of the applications during their priming runs; our kernel shows no overhead even when the kernel must initiate I/O to retrieve data off the physical disk.

Microbenchmark Performance To better understand the different performance behaviors of the applications, we used microbenchmarks to measure the overhead our system introduces for primitive kernel operations. For these experiments, we configured HBench-OS to run each test 50 times.

Our results for basic system calls (Table 5) indicate that the original SVA system adds significant overhead (on the order of tens of microseconds) to individual system calls. However, the results also show that our new safety checks only add a small amount of additional overhead (25% or less) to the original SVA system.

We also tested the file system bandwidth, shown in Figure 4. The results show that the original SVA system reduces file system bandwidth by about 5-20% for small files but that the overhead for larger files is negligible. Again, however, the additional checks for low-level kernel operations add no overhead.

The microbenchmark results provide a partial explanation for the application performance results. The applications in Table 3 experience no overhead because they perform most of their processing in user-space; the overhead of the kernel does not affect them much. In contrast, the `sshd` and `thttpd` servers spend most of their time executing in the kernel (primarily in the `poll()`, `select()`, and `write()` system calls). For the system calls that we tested, our new safety checks add less than several microseconds of overhead (as shown in Table 5). For a small network transfer of 1 KB (which takes less than 8 μ s on a Gigabit network), such an overhead can affect performance. However, for larger file sizes (e.g., an 8 MB transfer that takes 62.5 ms), this overhead becomes negligible. This effect shows up in our results for networked applications (`thttpd` and `sshd`): smaller file transfers see significant overhead, but past a certain file size, the overhead from the run-time safety checks becomes negligible.

Benchmark	i386 (μ s)	SVA (μ s)	SVA-OS (μ s)	% Increase from SVA to SVA-OS	Description
getpid	0.16 (0.001)	0.37 (0.000)	0.37 (0.006)	0.0%	Latency of getpid() syscall
open/close	1.10 (0.009)	11.1 (0.027)	12.1 (0.076)	9.0%	Latency of opening and closing a file
write	0.25 (0.001)	1.87 (0.012)	1.86 (0.010)	-0.4%	Latency of writing a single byte to /dev/null
signal handler	1.59 (0.006)	6.88 (0.044)	8.49 (0.074)	23%	Latency of calling a signal handler
signal install	0.34 (0.001)	1.56 (0.019)	1.95 (0.007)	25%	Latency of installing a signal handler
pipe latency	2.74 (0.014)	30.5 (0.188)	35.9 (0.267)	18%	Latency of ping-ponging one byte message between two processes
poll	1.16 (0.043)	6.47 (0.080)	7.03 (0.014)	8.7%	Latency of polling both ends of a pipe for reading and writing. Data is always available for reading.
select	1.00 (0.019)	8.18 (0.133)	8.81 (0.020)	7.7%	Latency of testing both ends of a pipe for reading and writing. Data is always available for reading.

Table 5: Latency of Kernel Operations. Standard Deviation Shown in Parentheses.

8 Related Work

Previous work has explored several approaches to providing greater safety and reliability for operating system kernels. Some require complete OS re-design, e.g., capability-based operating systems [37, 38] and micro-kernels [1, 25]. Others use isolation (or “sandboxing”) techniques, including device driver isolation within the OS [35, 44, 45, 51] or the hypervisor [17]. While effective at increasing system reliability, none of these approaches provide the memory safety guarantees provided by our system, e.g., none of these prevent corruption of memory mapped I/O devices, unsafe context switching, or improper configuration of the MMU by either kernel or device driver code. In fact, none of these approaches could protect against the Linux exploits or device corruption cases described in Section 7. In contrast, our system offers protection from all of these problems for both driver code and core kernel code.

The EROS [38] and Coyotos [37] systems provide a form of safe (dynamic) typing for abstractions, e.g., capabilities, at their higher-level OS (“node and page”) layer. This type safety is preserved throughout the design, even across I/O operations. The lower-level layer, which implements these abstractions, is written in C/C++ and is theoretically vulnerable to memory safety errors but is designed carefully to minimize them. The design techniques used here are extremely valuable but difficult to retrofit to commodity systems.

Some OSs written in type-safe languages, including JX [18], SPIN [21], Singularity [22], and others [20] provide abstractions that guarantee that loads and stores to I/O devices do not access main memory, and main memory accesses do not access I/O device memory. However, these systems either place context switching and MMU management within the virtual machine run-time (JX) or provide no guarantee that errors in these operations cannot compromise the safety guarantees of the language in which they are written.

Another approach that could provide some of the guar-

antees of our work is to add annotations to the C language. For example, SafeDrive’s annotation system [51] could be extended to provide our I/O memory protections and perhaps some of our other safety guarantees. Such an approach, however, would likely require changes to every driver and kernel module, whereas our approach only requires a one-time port to the SVA instruction set and very minor changes to machine-independent parts of the kernel.

The Devil project [27] defines a safe interface to hardware devices that enforces safety properties. Devil could ensure that writes to the device’s memory did not access kernel memory, but not vice versa. Our SVA extensions also protect I/O memory from kernel memory and provide comprehensive protection for other low-level hardware interactions, such as MMU changes, context switching, and thread management.

Mondrix [49] provides isolation between memory spaces within a kernel using a word-granularity memory isolation scheme implemented in hardware [48]. Because Mondrix enables much more fine-grained isolation (with acceptable overhead) than the software supported isolation schemes discussed earlier, it may be able to prevent some or all of the memory-related exploits we discuss. Nevertheless, it cannot protect against other errors such as control flow violations or stack manipulation.

A number of systems provide Dynamic Information Flow Tracking or “taint tracking” to enforce a wide range of security policies, including memory safety, but most of these have only reported results for user-space applications. Raksha [12] employed fine-grain information flow policies, supported by special hardware, to prevent buffer overflow attacks on the Linux kernel by ensuring that injected values weren’t dereferenced as pointers. Unlike our work, it does not protect against attacks that inject non-pointer data nor does it prevent use-after-free errors of kernel stacks and other state buffers used in low-level kernel/hardware interaction. Furthermore, this system does not work on commodity hardware.

The CacheKernel [7] partitions its functionality into an application-specific OS layer and a common “cache kernel” that handles context-switching, memory mappings, etc. The CacheKernel does not aim to provide memory safety, but its two layers are conceptually similar to the commodity OS and the virtual machine in our approach. A key design difference, however, is that our interface also attempts to make kernel code easier to analyze. For example, state manipulation for interrupted programs is no longer an arbitrary set of loads/stores to memory but a single instruction with a semantic meaning.

Our system employs techniques from VMMs. The API provided by SVA for configuring the MMU securely is similar to that presented by para-virtualized hypervisors [16, 50]. However, unlike VMMs, our use of these mechanisms is to provide fine-grain protection internal to a single domain, including isolation between user and kernel space and protection of type-safe main memory, saved processor state, and the kernel stack. For example, hypervisors would not be able to guard against [42], which our system does prevent, even though it is an MMU error. Also, a hypervisor that uses binary rewriting internally, e.g., for instrumenting itself, could be vulnerable to [9], just as the Linux kernel was. We believe VMMs could be a useful *target* for our work.

SecVisor [36] is a hypervisor that ensures that only approved code is executed in the processor’s privileged mode. In contrast, our system does not ensure that kernel code meets a set of requirements other than being memory safe. Unlike SVA, SecVisor does not ensure that the approved kernel code is memory safe.

9 Conclusion

In this paper, we have presented new mechanisms to ensure that low-level kernel operations such as processor state manipulation, stack management, memory mapped I/O, MMU updates, and self-modifying code do not violate the assumptions made by memory safety checkers. We implemented our design in the Secure Virtual Architecture (SVA) system, a safe execution environment for commodity operating systems, and its corresponding port of Linux 2.4.22. Only around 100 lines of code were added or changed to the SVA-port Linux kernel for the new techniques. To our knowledge, this is the first paper that (i) describes a design to prevent bugs in low-level kernel operations from compromising memory safe operating systems, including operating systems written in safe or unsafe languages; and (ii) implements and evaluates a system that guards against such errors.

Our experiments show that the additional runtime checks add little overhead to the original SVA prototype and were able to catch multiple real-world exploits that would otherwise bypass the memory safety guarantees

provided by the original SVA system. Taken together, these results indicate that it is clearly worthwhile to add these techniques to an existing memory safety system.

Acknowledgments

We wish to thank our shepherd, Trent Jaeger, and the anonymous reviewers for their helpful and insightful feedback.

References

- [1] ACCETTA, M., BARON, R., BOLOSKY, W., GOLUB, D., RASHID, R., TEVANIAN, A., AND YOUNG, M. Mach: A new kernel foundation for unix development. In *Proc. USENIX Annual Technical Conference* (Atlanta, GA, USA, July 1986), pp. 93–113.
- [2] APPLE COMPUTER, INC. Apple Mac OS X kernel semop local stack-based buffer overflow vulnerability, April 2005. <http://www.securityfocus.com/bid/13225>.
- [3] ARCANGELI, A. Linux kernel mremap local privilege escalation vulnerability, May 2006. <http://www.securityfocus.com/bid/18177>.
- [4] BERSHAD, B., SAVAGE, S., PARDYAK, P., SIRER, E. G., BECKER, D., FIUCZYNSKI, M., CHAMBERS, C., AND EGGERS, S. Extensibility, Safety and Performance in the SPIN Operating System. In *Proc. ACM SIGOPS Symp. on Op. Sys. Principles* (Copper Mountain, CO, USA, 1995), pp. 267–284.
- [5] BOVET, D. P., AND CESATI, M. *Understanding the LINUX Kernel*, 2nd ed. O’Reilly, Sebastopol, CA, 2003.
- [6] BROWN, A. *A Decompositional Approach to Computer System Performance*. PhD thesis, Harvard College, April 1997.
- [7] CHERITON, D. R., AND DUDA, K. J. A caching model of operating system kernel functionality. In *Proc. USENIX Symp. on Op. Sys. Design and Impl.* (Monterey, CA, USA, November 1994), pp. 179–193.
- [8] CORBET. SMP alternatives, December 2005. <http://lwn.net/Articles/164121>.
- [9] CORBET, J. The source of the e1000e corruption bug, October 2008. <http://lwn.net/Articles/304105>.
- [10] CRISWELL, J., LENHART, A., DHURJATI, D., AND ADVE, V. Secure Virtual Architecture: A Safe Execution Environment for Commodity Operating Systems. In *Proc. ACM SIGOPS Symp. on Op. Sys. Principles* (Stevenson, WA, USA, October 2007), pp. 351–366.
- [11] CRISWELL, J., MONROE, B., AND ADVE, V. A virtual instruction set interface for operating system kernels. In *Workshop on the Interaction between Operating Systems and Computer Architecture* (Boston, MA, USA, June 2006), pp. 26–33.
- [12] DALTON, M., KANNAN, H., AND KOZYRAKIS, C. Real-world buffer overflow protection for userspace & kernelspace. In *Proceedings of the USENIX Security Symposium* (San Jose, CA, USA, 2008), pp. 395–410.
- [13] DEVICES, A. M. AMD64 architecture programmer’s manual volume 2: System programming, September 2006.
- [14] DHURJATI, D., AND ADVE, V. Backwards-compatible array bounds checking for C with very low overhead. In *Proc. of the Int’l Conf. on Software Engineering* (Shanghai, China, May 2006), pp. 162–171.

- [15] DHURJATI, D., KOWSHIK, S., AND ADVE, V. SAFECode: Enforcing alias analysis for weakly typed languages. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)* (Ottawa, Canada, June 2006), pp. 144–157.
- [16] DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., PRATT, I., WARFIELD, A., BARHAM, P., AND NEUGEBAUER, R. Xen and the art of virtualization. In *Proc. ACM SIGOPS Symp. on Op. Sys. Principles* (Bolton Landing, NY, USA, October 2003), pp. 164–177.
- [17] FRASER, K., HAND, S., NEUGEBAUER, R., PRATT, I., WARFIELD, A., AND WILLIAMS, M. Safe hardware access with the xen virtual machine monitor. In *Proceedings of the First Workshop on Operating System and Architectural Support for the on demand IT Infrastructure* (Boston, MA, USA, October 2004).
- [18] GOLM, M., FELSER, M., WAWERSICH, C., AND KLEINODER, J. The JX Operating System. In *Proc. USENIX Annual Technical Conference* (Monterey, CA, USA, June 2002), pp. 45–58.
- [19] GUNINSKI, G. Linux kernel multiple local vulnerabilities, 2005. <http://www.securityfocus.com/bid/11956>.
- [20] HALLGREN, T., JONES, M. P., LESLIE, R., AND TOLMACH, A. A principled approach to operating system construction in Haskell. In *Proc. ACM SIGPLAN Int'l Conf. on Functional Programming* (Tallin, Estonia, September 2005), pp. 116–128.
- [21] HSIEH, W., FIUCZYNSKI, M., GARRETT, C., SAVAGE, S., BECKER, D., AND BERSHAD, B. Language support for extensible operating systems. In *Workshop on Compiler Support for System Software* (Arizona, USA, February 1996).
- [22] HUNT, G. C., LARUS, J. R., ABADI, M., AIKEN, M., BARHAM, P., FHNDRIKH, M., HODSON, C. H. O., LEVI, S., MURPHY, N., STEENSGAARD, B., TARDITI, D., WOBBER, T., AND ZILL, B. An overview of the Singularity project. Tech. Rep. MSR-TR-2005-135, Microsoft Research, October 2005.
- [23] LATTNER, C., AND ADVE, V. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proc. Conf. on Code Generation and Optimization* (San Jose, CA, USA, Mar 2004), pp. 75–88.
- [24] LATTNER, C., LENHARTH, A. D., AND ADVE, V. S. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)* (San Diego, CA, USA, June 2007), pp. 278–289.
- [25] LIEDTKE, J. On micro-kernel construction. *SIGOPS Oper. Syst. Rev.* 29, 5 (1995), 237–250.
- [26] LMH. Month of kernel bugs (MoKB) archive, 2006. <http://projects.info-pull.com/mokb/>.
- [27] MÉRILLON, F., RÉVEILLÈRE, L., CONSEL, C., MARLET, R., AND MULLER, G. Devil: an IDL for hardware programming. In *USENIX Symposium on Operating System Design and Implementation* (San Diego, CA, USA, October 2000), pp. 17–30.
- [28] MICROSYSTEMS, S. Sun solaris sysinfo system call kernel memory reading vulnerability, October 2003. <http://www.securityfocus.com/bid/8831>.
- [29] MONROE, B. M. Measuring and improving the performance of Linux on a virtual instruction set architecture. Master's thesis, Computer Science Dept., Univ. of Illinois at Urbana-Champaign, Urbana, IL, Dec 2005.
- [30] NECULA, G. C., CONDIT, J., HARREN, M., MCPEAK, S., AND WEIMER, W. Ccured: type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems* (2005).
- [31] POSKANZE, J. thtpd - tiny/turbo/throttling http server, 2000. <http://www.acme.com/software/thtpd>.
- [32] RUWASE, O., AND LAM, M. A practical dynamic buffer overflow detector. In *In Proceedings of the Network and Distributed System Security (NDSS) Symposium* (San Diego, CA, USA, 2004), pp. 159–169.
- [33] SAULPAUGH, T., AND MIRHO, C. *Inside the JavaOS Operating System*. Addison-Wesley, Reading, MA, USA, 1999.
- [34] SCOTT, M. L. *Programming Language Pragmatics*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 2001.
- [35] SELTZER, M. I., ENDO, Y., SMALL, C., AND SMITH, K. A. Dealing with disaster: Surviving misbehaved kernel extensions. In *USENIX Symposium on Operating System Design and Implementation* (Seattle, WA, October 1996), pp. 213–227.
- [36] SESHADRI, A., LUK, M., QU, N., AND PERRIG, A. Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. *SIGOPS Oper. Syst. Rev.* 41, 6 (2007), 335–350.
- [37] SHAPIRO, J., DOERRIE, M. S., NORTHUP, E., SRIDHAR, S., AND MILLER, M. Towards a verified, general-purpose operating system kernel. In *1st NICTA Workshop on Operating System Verification* (Sydney, Australia, October 2004).
- [38] SHAPIRO, J. S., AND ADAMS, J. Design evolution of the EROS single-level store. In *Proceedings of the USENIX Annual Technical Conference* (Berkeley, CA, USA, June 2002), pp. 59–72.
- [39] STARSETZ, P. Linux kernel do_mremap function vma limit local privilege escalation vulnerability, February 2004. <http://www.securityfocus.com/bid/9686>.
- [40] STARZETZ, P. Linux kernel elf core dump local buffer overflow vulnerability. <http://www.securityfocus.com/bid/13589>.
- [41] STARZETZ, P. Linux kernel IGMP multiple vulnerabilities, 2004. <http://www.securityfocus.com/bid/11917>.
- [42] STARZETZ, P., AND PURCZYNSKI, W. Linux kernel do_mremap function boundary condition vulnerability, January 2004. <http://www.securityfocus.com/bid/9356>.
- [43] STARZETZ, P., AND PURCZYNSKI, W. Linux kernel setsockopt MCAST_MSFILTER integer overflow vulnerability, 2004. <http://www.securityfocus.com/bid/10179>.
- [44] SWIFT, M. M., BERSHAD, B. N., AND LEVY, H. M. Improving the reliability of commodity operating systems. *ACM Trans. Comput. Syst.* 23, 1 (2005), 77–110.
- [45] ÚLFAR ERLINGSSON, ABADI, M., VRABLE, M., BUDI, M., AND NECULA, G. C. XFI: Software guards for system address spaces. In *USENIX Symposium on Operating System Design and Implementation* (Seattle, WA, USA, November 2006), pp. 75–88.
- [46] VAN SPRUNDEL, I. Linux kernel bluetooth signed buffer index vulnerability. <http://www.securityfocus.com/bid/12911>.
- [47] WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review* 27, 5 (1993), 203–216.
- [48] WITCHEL, E., CATES, J., AND ASANOVIC, K. Mondrian memory protection. In *Proc. Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (San Jose, CA, USA, October 2002), pp. 304–316.
- [49] WITCHEL, E., RHEE, J., AND ASANOVIC, K. Mondrix: Memory isolation for linux using mondriaan memory protection. In *Proc. ACM SIGOPS Symp. on Op. Sys. Principles* (Brighton, UK, October 2005), pp. 31–44.
- [50] WRIGHT, C. Para-virtualization interfaces, 2006. <http://lwn.net/Articles/194340>.
- [51] ZHOU, F., CONDIT, J., ANDERSON, Z., BAGRAK, I., ENNALS, R., HARREN, M., NECULA, G., AND BREWER, E. Safedrive: Safe and recoverable extensions using language-based techniques. In *USENIX Symposium on Operating System Design and Implementation* (Seattle, WA, USA, November 2006), pp. 45–60.

Detecting Spammers with SNARE: Spatio-temporal Network-level Automatic Reputation Engine

Shuang Hao, Nadeem Ahmed Syed, Nick Feamster, Alexander G. Gray, Sven Krasser *
College of Computing, Georgia Tech *McAfee, Inc.
{shao, nadeem, feamster, agray}@cc.gatech.edu, sven_krasser@mcafee.com

Abstract

Users and network administrators need ways to filter email messages based primarily on the reputation of the sender. Unfortunately, conventional mechanisms for sender reputation—notably, IP blacklists—are cumbersome to maintain and evadable. This paper investigates ways to infer the reputation of an email sender based solely on network-level features, without looking at the contents of a message. First, we study first-order properties of network-level features that may help distinguish spammers from legitimate senders. We examine features that can be ascertained without ever looking at a packet’s contents, such as the distance in IP space to other email senders or the geographic distance between sender and receiver. We derive features that are *lightweight*, since they do not require seeing a large amount of email from a single IP address and can be gleaned without looking at an email’s contents—many such features are apparent from even a single packet. Second, we incorporate these features into a classification algorithm and evaluate the classifier’s ability to automatically classify email senders as spammers or legitimate senders. We build an *automated* reputation engine, *SNARE*, based on these features using labeled data from a deployed commercial spam-filtering system. We demonstrate that *SNARE* can achieve comparable accuracy to existing static IP blacklists: about a 70% detection rate for less than a 0.3% false positive rate. Third, we show how *SNARE* can be integrated into existing blacklists, essentially as a first-pass filter.

1 Introduction

Spam filtering systems use two mechanisms to filter spam: content filters, which classify messages based on the contents of a message; and sender reputation, which maintains information about the IP address of a sender as an input to filtering. Content filters (e.g., [22, 23])

can block certain types of unwanted email messages, but they can be brittle and evadable, and they require analyzing the contents of email messages, which can be expensive. Hence, spam filters also rely on *sender reputation* to filter messages; the idea is that a mail server may be able to reject a message purely based on the reputation of the sender, rather than the message contents. DNS-based blacklists (DNSBLs) such as Spamhaus [7] maintain lists of IP addresses that are known to send spam. Unfortunately, these blacklists can be both incomplete and slow-to-respond to new spammers [32]. This unresponsiveness will only become more serious as both botnets and BGP route hijacking make it easier for spammers to dynamically obtain new, unlisted IP addresses [33, 34]. Indeed, network administrators are still searching for spam-filtering mechanisms that are both *lightweight* (i.e., they do not require detailed message or content analysis) and *automated* (i.e., they do not require manual update, inspection, or verification).

Towards this goal, this paper presents *SNARE* (Spatio-temporal Network-level Automatic Reputation Engine), a sender reputation engine that can accurately and automatically classify email senders based on lightweight, network-level features that can be determined early in a sender’s history—sometimes even upon seeing only a single packet. *SNARE* relies on the intuition that about 95% of all email is spam, and, of this, 75 – 95% can be attributed to botnets, which often exhibit unusual sending patterns that differ from those of legitimate email senders. *SNARE* classifies senders based on *how* they are sending messages (i.e., traffic patterns), rather than *who* the senders are (i.e., their IP addresses). In other words, *SNARE* rests on the assumption that there are lightweight network-level features that can differentiate spammers from legitimate senders; this paper finds such features and uses them to build a system for automatically determining an email sender’s reputation.

SNARE bears some similarity to other approaches that classify senders based on network-level behavior [12, 21,

24, 27, 34], but these approaches rely on inspecting the message contents, gathering information across a large number of recipients, or both. In contrast, *SNARE* is based on *lightweight* network-level features, which could allow it to scale better and also to operate on higher traffic rates. In addition, *SNARE* is *more accurate* than previous reputation systems that use network-level behavioral features to classify senders: for example, *SNARE*'s false positive rate is an order of magnitude less than that in our previous work [34] for a similar detection rate. It is the first reputation system that is both as accurate as existing static IP blacklists and automated to keep up with changing sender behavior.

Despite the advantages of automatically inferring sender reputation based on “network-level” features, a major hurdle remains: We must identify *which features* effectively and efficiently distinguish spammers from legitimate senders. Given the massive space of possible features, finding a collection of features that classifies senders with both low false positive and low false negative rates is challenging. This paper identifies thirteen such network-level features that require varying levels of information about senders' history.

Different features impose different levels of overhead. Thus, we begin by evaluating features that can be computed purely locally at the receiver, with no information from other receivers, no previous sending history, and no inspection of the message itself. We found several features that fall into this category are surprisingly effective for classifying senders, including: The AS of the sender, the geographic distance between the IP address of the sender and that of the receiver, the density of email senders in the surrounding IP address space, and the time of day the message was sent. We also looked at various aggregate statistics across messages and receivers (e.g., the mean and standard deviations of messages sent from a single IP address) and found that, while these features require slightly more computation and message overhead, they do help distinguish spammers from legitimate senders as well. After identifying these features, we analyze the relative importance of these features and incorporate them into an automated reputation engine, based on the *RuleFit* [19] ensemble learning algorithm.

In addition to presenting the first automated classifier based on network-level features, this paper presents several additional contributions. First, we presented a detailed study of various network-level characteristics of both spammers and legitimate senders, a detailed study of how well each feature distinguishes spammers from legitimate senders, and explanations of why these features are likely to exhibit differences between spammers and legitimate senders. Second, we use state-of-the-art ensemble learning techniques to build a classifier using these features. Our results show that *SNARE*'s perfor-

mance is at least as good as static DNS-based blacklists, achieving a 70% detection rate for about a 0.2% false positive rate. Using features extracted from a single message and aggregates of these features provides slight improvements, and adding an AS “whitelist” of the ASes that host the most commonly misclassified senders reduces the false positive rate to 0.14%. This accuracy is roughly equivalent to that of existing static IP blacklists like SpamHaus [7]; the advantage, however, is that *SNARE* is *automated*, and it characterizes a sender based on its sending *behavior*, rather than its IP address, which may change due to dynamic addressing, newly compromised hosts, or route hijacks. Although *SNARE*'s performance is still not perfect, we believe that the benefits are clear: Unlike other email sender reputation systems, *SNARE* is both automated and lightweight enough to operate solely on network-level information. Third, we provide a deployment scenario for *SNARE*. Even if others do not deploy *SNARE*'s algorithms exactly as we have described, we believe that the collection of network-level features themselves may provide useful inputs to other commercial and open-source spam filtering appliances.

The rest of this paper is organized as follows. Section 2 presents background on existing sender reputation systems and a possible deployment scenario for *SNARE* and introduces the ensemble learning algorithm. Section 3 describes the network-level behavioral properties of email senders and measures first-order statistics related to these features concerning both spammers and legitimate senders. Section 4 evaluates *SNARE*'s performance using different feature subsets, ranging from those that can be determined from a single packet to those that require some amount of history. We investigate the potential to incorporate the classifier into a spam-filtering system in Section 5. Section 6 discusses evasion and other limitations, Section 7 describes related work, and Section 8 concludes.

2 Background

In this section, we provide background on existing sender reputation mechanisms, present motivation for improved sender reputation mechanisms (we survey other related work in Section 7), and describe a classification algorithm called *RuleFit* to build the reputation engine. We also describe McAfee's TrustedSource system, which is both the source of the data used for our analysis and a possible deployment scenario for *SNARE*.

2.1 Email Sender Reputation Systems

Today's spam filters look up IP addresses in DNS-based blacklists (DNSBLs) to determine whether an IP address is a known source of spam at the time

field is the IP address being queried (i.e., the IP address of the email sender). The IP addresses of the senders are shown in the Hilbert space, as in Figure 2¹, where each pixel represents a /24 network prefix and the intensity indicates the observed IP density in each block. The distribution of the senders' IP addresses shows that the TrustedSource database collocated a representative set of email across the Internet. We use many of the other features in Figure 1 as input to *SNARE*'s classification algorithms.

To help us label senders as either spammers or legitimate senders for both our feature analysis (Section 3) and training (Sections 2.3 and 4), the logs also contain *scores* for each email message that indicate how McAfee scored the email sender based on its current system. The `score` field indicates McAfee's sender reputation score, which we stratify into five labels: certain ham, likely ham, certain spam, likely ham, and uncertain. Although these scores are not perfect ground truth, they do represent the output of both manual classification and continually tuned algorithms that also operate on more heavy-weight features (e.g., packet payloads). Our goal is to develop a fully automated classifier that is as accurate as TrustedSource but (1) classifies senders *automatically* and (2) relies only on lightweight, evasion-resistant network-level features.

Deployment and data aggregation scenario Because it operates only on network-level features of email messages, *SNARE* could be deployed either as part of TrustedSource or as a standalone DNSBL. Some of the features that *SNARE* uses rely on aggregating sender behavior across a wide variety of senders. To aggregate these features, a monitor could collect information about the global behavior of a sender across a wide variety of recipient domains. Aggregating this information is a reasonably lightweight operation: Since the features that *SNARE* uses are based on simple features (i.e., the IP address, plus auxiliary information), they can be piggy-backed in small control messages or in DNS messages (as with McAfee's TrustedSource deployment).

2.3 Supervised Learning: RuleFit

Ensemble learning: *RuleFit* Learning ensembles have been among the popular predictive learning methods over the last decade. Their structural model takes the form

$$F(\mathbf{x}) = a_0 + \sum_{m=1}^M a_m f_m(\mathbf{x}) \quad (1)$$

Where \mathbf{x} are input variables derived from the training data (spatio-temporal features); $f_m(\mathbf{x})$ are different

¹A larger figure is available at <http://www.gtnoise.net/snare/hilbert-ip.png>.

functions called ensemble members ("base learner") and M is the size of the ensemble; and $F(\mathbf{x})$ is the predictive output (labels for "spam" or "ham"), which takes a linear combination of ensemble members. Given the base learners, the technique determines the parameters for the learners by regularized linear regression with a "lasso" penalty (to penalize large coefficients a_m).

Friedman and Popescu proposed *RuleFit* [19] to construct regression and classification problems as linear combinations of simple rules. Because the number of base learners in this case can be large, the authors propose using the rules in a decision tree as the base learners. Further, to improve the accuracy, the variables themselves are also included as basis functions. Moreover, fast algorithms for minimizing the loss function [18] and the strategy to control the tree size can greatly reduce the computational complexity.

Variable importance Another advantage of *RuleFit* is the interpretation. Because of its simple form, each rule is easy to understand. The relative importance of the respective variables can be assessed after the predictive model is built. Input variables that frequently appear in important rules or basic functions are deemed more relevant. The importance of a variable x_i is given as importance of the basis functions that correspond directly to the variable, plus the average importance of all the other rules that involve x_i . The *RuleFit* paper has more details [19]. In Section 4.3, we show the relative importance of these features.

Comparison to other algorithms There exist two other classic classifier candidates, both of which we tested on our dataset and both of which yielded poorer performance (i.e., higher false positive and lower detection rates) than *RuleFit*. Support Vector Machine (SVM) [15] has been shown empirically to give good generalization performance on a wide variety of problems such as handwriting recognition, face detection, text categorization, etc. On the other hand, they do require significant parameter tuning before the best performance can be obtained. If the training set is large, the classifier itself can take up a lot of storage space and classifying new data points will be correspondingly slower since the classification cost is $O(S)$ for each test point, where S is the number of support vectors. The computational complexity of SVM conflicts with *SNARE*'s goal to make decision quickly (at line rate). Decision trees [30] are another type of popular classification method. The resulting classifier is simple to understand and faster, with the prediction on a new test point taking $O(\log(N))$, where N is the number of nodes in the trained tree. Unfortunately, decision trees compromise accuracy: its high false positive rates make it less than ideal for our purpose.

3 Network-level Features

In this section, we explore various spatio-temporal features of email senders and discuss why these properties are relevant and useful for differentiating spammers from legitimate senders. We categorize the features we analyze by increasing level of overhead:

- *Single-packet features* are those that can be determined with no previous history from the IP address that *SNARE* is trying to classify, and given only a *single packet* from the IP address in question (Section 3.1).
- *Single-header and single-message features* can be gleaned from a single SMTP message header or email message (Section 3.2).
- *Aggregate features* can be computed with varying amounts of history (i.e., aggregates of other features) (Section 3.3).

Each class of features contains those that may be either purely local to a single receiver or aggregated across multiple receivers; the latter implies that the reputation system must have some mechanism for aggregating features in the network. In the following sections, we describe features in each of these classes, explain the intuition behind selecting that feature, and compare the feature in terms of spammers vs. legitimate senders.

No single feature needs to be perfectly discriminative between ham and spam. The analysis below shows that it is unrealistic to have a single perfect feature to make optimal resolution. As we describe in Section 2.3, *SNARE*'s classification algorithm uses a *combination* of these features to build the best classifier. We do, however, evaluate *SNARE*'s classifier using these three different classes of features to see how well it can perform using these different classes. Specifically, we evaluate how well *SNARE*'s classification works using only single-packet features to determine how well such a lightweight classifier would perform; we then see whether using additional features improves classification.

3.1 Single-Packet Features

In this section, we discuss some properties for identifying a spammer that rely only on a single packet from the sender IP address. In some cases, we also rely on auxiliary information, such as routing table information, sending history from neighboring IP addresses, etc., not solely information in the packet itself. We first discuss the features that can be extracted from just a single IP packet: the geodesic distance between the sender and receiver, sender neighborhood density, probability ratio of spam to ham at the time-of-day the IP packet arrives, AS number of the sender and the status of open ports on the

machine that sent the email. The analysis is based on the McAfee's data from October 22–28, 2007 inclusive (7 days).²

3.1.1 Sender-receiver geodesic distance: Spam travels further

Recent studies suggest that social structure between communicating parties could be used to effectively isolate spammers [13, 20]. Based on the findings in these studies, we hypothesized that legitimate emails tend to travel shorter geographic distances, whereas the distance traveled by spam will be closer to random. In other words, a spam message may be just as likely to travel a short distance as across the world.

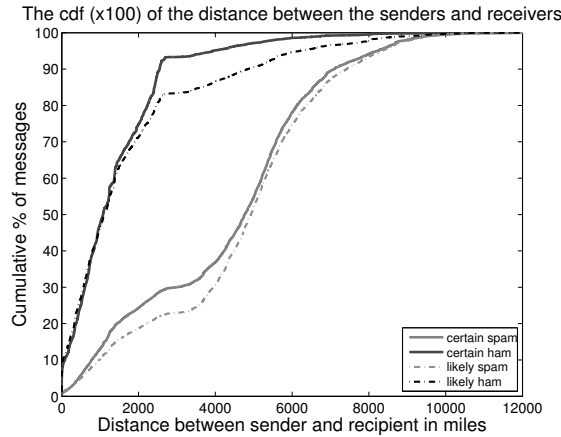
Figure 3(a) shows that our intuition is roughly correct: the distribution of the distance between the sender and the target IP addresses for each of the four categories of messages. The distance used in these plots is the geodesic distance, that is, the distance along the surface of the earth. It is computed by first finding the physical latitude and longitude of the source and target IP using the MaxMind's GeoIP database [8] and then computing the distance between these two points. These distance calculations assume that the earth is a perfect sphere. For *certain ham*, 90% of the messages travel about 2,500 miles or less. On the other hand, for *certain spam*, only 28% of messages stay within this range. In fact, about 10% of spam travels more than 7,000 miles, which is a quarter of the earth's circumference at the equator. These results indicate that geodesic distance is a promising metric for distinguishing spam from ham, which is also encouraging, since it can be computed quickly using just a single IP packet.

3.1.2 Sender IP neighborhood density: Spammers are surrounded by other spammers

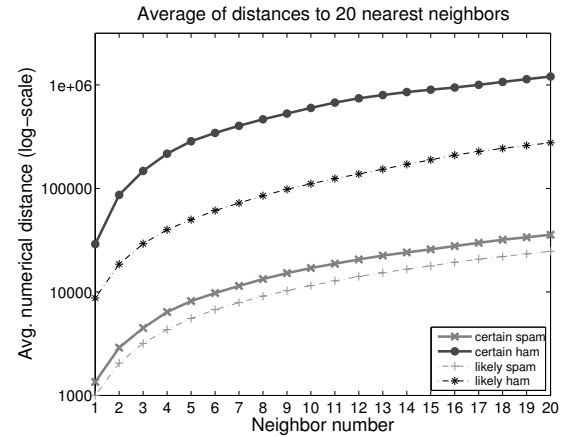
Most spam messages today are generated by botnets [33, 37]. For messages originating from the same botnet, the infected IP addresses may all lie close to one another in numerical space, often even within the same subnet. One way to detect whether an IP address belongs to a botnet is to look at the past history and determine if messages have been received from other IPs in the same subnet as the current sender, where the subnet size can be determined experimentally. If many different IPs from the same subnet are sending email, the likelihood that the whole subnet is infested with bots is high.

The problem with simply using subnet density is that the frame of reference does not transcend the subnet

²The evaluation in Section 4 uses the data from October 22–November 4, 2007 (14 days), some of which are not included in the data trace used for measurement study.



(a) Geodesic distance between the sender and recipient's geographic location.



(b) Average of numerical distances to the 20 nearest neighbors in the IP space.

Figure 3: Spatial differences between spammers and legitimate senders.

boundaries. A more flexible measure of *email sender density* in an IP's neighborhood is the distances to its k nearest neighbors. The distance to the k nearest neighbors can be computed by treating the IPs as set of numbers from 0 to $2^{32} - 1$ (for IPv4) and finding the nearest neighbors in this single dimensional space. We can expect these distances to exhibit different patterns for spam and ham. If the neighborhood is *crowded*, these neighbor distances will be small, indicating the possible presence of a botnet. In normal circumstances, it would be unusual to see a large number of IP addresses sending email in a small IP address space range (one exception might be a cluster of outbound mail servers, so choosing a proper threshold is important, and an operator may need to evaluate which threshold works best on the specific network where *SNARE* is running).

The average distances to the 20 nearest neighbors of the senders are shown in Figure 3(b). The x-axis indicates how many nearest neighbors we consider in IP space, and the y-axis shows the average distance in the sample to that many neighbors. The figure reflects the fact that a large majority of spam originates from hosts have high email sender density in a given IP region. The distance to the k^{th} nearest neighbor for spam tends to be much shorter on average than it is for legitimate senders, indicating that spammers generally reside in areas with higher densities of email senders (in terms of IP address space).

3.1.3 Time-of-day: Spammers send messages according to machine off/on patterns

Another feature that can be extracted using information from a single packet is the time of day when the message was sent. We use the *local* time of day at the sender's physical location, as opposed to Coordinated

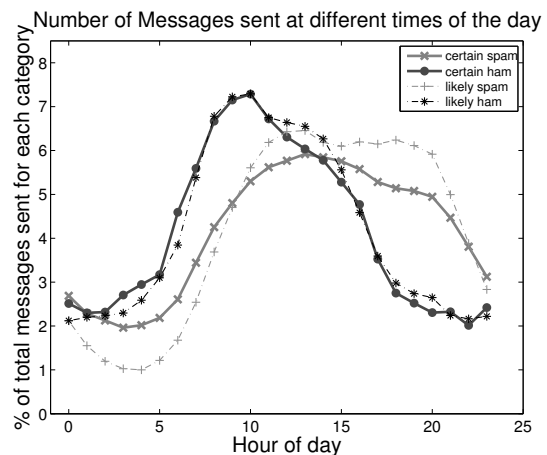


Figure 4: Differences in diurnal sending patterns of spammers and legitimate senders.

Universal Time (UTC). The intuition behind this feature is that local legitimate email sending patterns may more closely track “conventional” diurnal patterns, as opposed to spam sending patterns.

Figure 4 shows the relative percentage of messages of each type at different times of the day. The legitimate senders and the spam senders show different diurnal patterns. Two times of day are particularly striking: the relative amount of ham tends to ramp up quickly at the start of the workday and peaks in the early morning. Volumes decrease relatively quickly as well at the end of the workday. On the other hand spam increases at a slower, steadier pace, probably as machines are switched on in the morning. The spam volume stays steady throughout the day and starts dropping around 9:00 p.m., probably when machines are switched off again. In summary, legitimate

senders tend to follow workday cycles, and spammers tend to follow machine power cycles.

To use the timestamp as a feature, we compute the probability ratio of spam to ham at the time of the day when the message is received. First, we compute the *a priori* spam probability $p_{s,t}$ during some hour of the day t , as $p_{s,t} = n_{s,t}/n_s$, where $n_{s,t}$ is the number of spam messages received in hour t , and n_s is the number of spam messages received over the entire day. We can compute the *a priori* ham probability for some hour t , $p_{h,t}$ in a similar fashion. The probability ratio, r_t is then simply $p_{s,t}/p_{h,t}$. When a new message is received, the precomputed spam to ham probability ratio for the corresponding hour of the day at the senders timezone, r_t can be used as a feature; this ratio can be recomputed on a daily basis.

3.1.4 AS number of sender: A small number of ASes send a large fraction of spam

As previously mentioned, using IP addresses to identify spammers has become less effective for several reasons. First, IP addresses of senders are often transient. The compromised machines could be from dial-up users, which depend on dynamic IP assignment. If spam comes from mobile devices (like laptops), the IP addresses will be changed once the people carry the devices to a different place. In addition, spammers have been known to adopt stealthy spamming strategies where each bot only sends several spam to a single target domain, but overall the botnets can launch a huge amount of spam to many domains [33]. The low emission-rate and distributed attack requires to share information across domains for detection.

On the other hand, our previous study revealed that a significant portion of spammers come from a relatively small collection of ASes [33]. More importantly, the ASes responsible for spam differ from those that send legitimate email. As a result, the AS numbers of email senders could be a promising feature for evaluating the senders' reputation. Over the course of the seven days in our trace, more than 10% of unique spamming IPs (those sending certain spam) originated from only 3 ASes; the top 20 ASes host 42% of spamming IPs. Although our previous work noticed that a small number of ASes originated a large fraction of spam [33], we believe that this is the first work to suggest using the AS number of the email sender as input to an automated classifier for sender reputation.

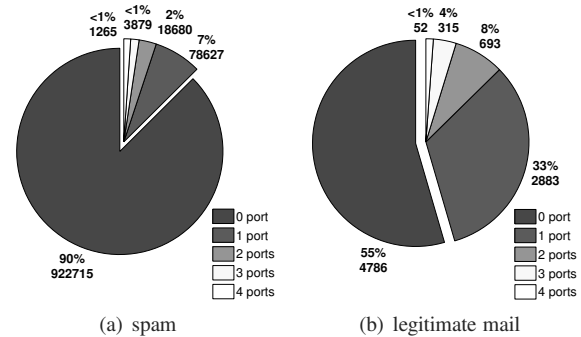


Figure 5: Distribution of number of open ports on hosts sending spam and legitimate mail.

3.1.5 Status of service ports: Legitimate mail tends to originate from machines with open ports

We hypothesized that legitimate mail senders may also listen on other ports besides the SMTP port, while bots might not; our intuition is that the bots usually send spam directly to the victim domain's mail servers, while the legitimate email is handed over from other domains' MSA (Mail Submission Agent). The techniques of reverse DNS (rDNS) and Forward Confirmed Reverse DNS (FCrDNS) have been widely used to check whether the email is from dial-up users or dynamically assigned addresses, and mail servers will refuse email from such sources [1].

We propose an additional feature that is orthogonal to DNSBL or rDNS checking. Outgoing mail servers open specific ports to accept users' connections, while the bots are compromised hosts, where the well-known service ports are closed (require root privilege to open). When packets reach the mail server, the server issues an active probe sent to the source host to scan the following four ports that are commonly used for outgoing mail service: 25 (SMTP), 465 (SSL SMTP), 80 (HTTP) and 443 (HTTPS), which are associated with outgoing mail services. Because neither the current mail servers nor the McAfee's data offer email senders' port information, we need to probe back sender's IP to check out what service ports might be open. The probe process was performed during both October 2008 and January 2009, well after the time when the email was received. Despite this delay, the status of open ports still exposes a striking difference between legitimate senders and spammers. Figure 5 shows the percentages and the numbers of opening ports for spam and ham categories respectively. The statistics are calculated on the senders' IPs from the evaluation dataset we used in Section 4 (October 22–28, 2007). In the spam case, 90% of spamming IP addresses have *none* of the standard mail service ports open; in contrast,

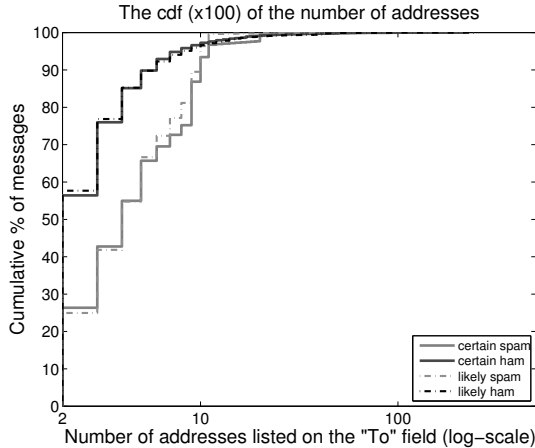


Figure 6: Distribution of number of addresses listed on the “To” field for each category (ignoring single-recipient messages).

half of the legitimate email comes from machines listening on at least one mail service port. Although firewalls might block the probing attempts (which causes the legitimate mail servers show no port listening), the status of the email-related ports still appears highly correlated with the distinction of the senders. When providing this feature as input to a classifier, we represent it as a bitmap (4 bits), where each bit indicates whether the sender IP is listening on a particular port.

3.2 Single-Header and Single-Message Features

In this section, we discuss other features that can be extracted from a single SMTP header or message: the number of recipients in the message, and the length of the message. We distinguish these features from those in the previous section, since extracting these features actually requires opening an SMTP connection, accepting the message, or both. Once a connection is accepted, and the SMTP header and subsequently, the complete message are received. At this point, a spam filter could extract additional non-content features.

3.2.1 Number of recipients: Spam tends to have more recipients

The features discussed so far can be extracted from a single IP packet from any given specific IP address combined with some historical knowledge of messages from other IPs. Another feature available without looking into the content is the number of address in “To” field of the header. This feature can be extracted after receiving the

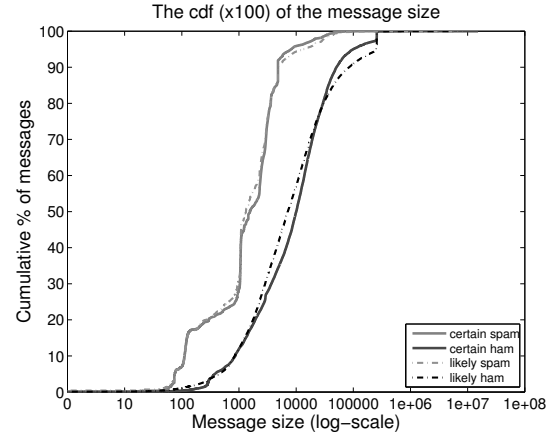


Figure 7: Distribution of message size (in bytes) for the different categories of messages.

entire SMTP header but before accepting the message body. However, the majority of messages only have one address listed. Over 94% of spam and 96% of legitimate email is sent to a single recipient. Figure 6 shows the distribution of number of addresses in the “To” field for each category of messages for all emails that are sent to more than one recipient. The x-axis is on a log-scale to focus the plot on the smaller values. Based on this plot and looking at the actual values, it appears that if there are very large number of recipients on the “To” field (100 or more), there does not seem to be a significant difference between the different types of senders for this measure. The noticeable differences around 2 to 10 addresses show that, generally, ham has fewer recipients (close to 2) while spam is sent to multiple addresses (close to 10). (We acknowledge that this feature is probably evadable and discuss this in more detail in Section 6.1).

3.2.2 Message size: Legitimate mail has variable message size; spam tends to be small

Once an entire message has been received, the email body size in bytes is also known. Because a given spam sender will mostly send the same or similar content in all the messages, it can be expected that the variance in the size of messages sent by a spammer will be lower than among the messages sent by a legitimate sender. To stay effective, the spam bots also need to keep the message size small so that they can maximize the number of messages they can send out. As such the spam messages can be expected to be biased towards the smaller size. Figure 7 shows the distribution of messages for each category. The spam messages are all clustered in the 1–10KB range, whereas the distribution of message size for legitimate senders is more evenly distributed. Thus, the mes-

sage body size is another property of messages that may help differentiate spammers from legitimate senders.

3.3 Aggregate Features

The behavioral properties discussed so far can all be constructed using a single message (with auxiliary or neighborhood information). If some history from an IP is available, some *aggregate IP-level features* can also be constructed. Given information about multiple messages from a single IP address, the overall *distribution* of the following measures can be captured by using a combination of *mean and variance of*: (1) geodesic distance between the sender and recipient, (2) number of recipients in the “To” field of the SMTP header, and (3) message body length in bytes. By summarizing behavior over multiple messages and over time, these aggregate features may yield a more reliable prediction. On the flip side, computing these features comes at the cost of increased latency as we need to collect a number of messages before we compute these. Sometimes gathering aggregate information even requires cross-domain collaboration. By averaging over multiple messages, these features may also smooth the structure of the feature space, making marginal cases more difficult to classify.

4 Evaluating the Reputation Engine

In this section, we evaluate the performance of *SNARE*’s *RuleFit* classification algorithm using different sets of features: those just from a single packet, those from a single header or message, and aggregate features.

4.1 Setup

For this evaluation, we used fourteen days of data from the traces, from October 22, 2007 to November 4, 2007, part of which are different from the analysis data in Section 3. In other words, the entire data trace is divided into two parts: the first half is used for measurement study, and the latter half is used to evaluate *SNARE*’s performance. The purpose of this setup is both to verify the hypothesis that the feature statistics we discovered would stick to the same distribution over time and to ensure that feature extraction would not interfere with our evaluation of prediction.

Training We first collected the features for each message for a subset of the trace. We then randomly sampled 1 million messages from each day on average, where the volume ratio of spam to ham is the same as the original data (i.e., 5% ham and 95% spam; for now, we consider only messages in the “certain ham” and “certain spam” categories to obtain more accurate ground truth). Only

our evaluation is based on this sampled dataset, *not* the feature analysis from Section 3, so the selection of those features should not have been affected by sampling. We then intentionally sampled equal amounts of spam as the ham data (30,000 messages in each categories for each day) to train the classifier because training requires that each class have an equal number of samples. In practice, spam volume is huge, and much spam might be discarded before entering the *SNARE* engine, so sampling on spam for training is reasonable.

Validation We evaluated the classifier using temporal cross-validation, which is done by splitting the dataset into subsets along the time sequence, training on the subset of the data in a time window, testing using the next subset, and moving the time window forward. This process is repeated ten times (testing on October 26, 2007 to November 4, 2007), with each subset accounting for one-day data and the time window set as 3 days (which indicates that long-period history is not required). For each round, we compute the detection rate and false positive rate respectively, where the detection rate (the “true positive” rate) is the ratio of spotted spam to the whole spam corpus, and false positive rate reflects the proportion of misclassified ham to all ham instances. The final evaluation reflects the average computed over all trials.

Summary Due to the high sampling rate that we used for this experiment, we repeated the above experiment for several trials to ensure that the results were consistent across trials. As the results in this section show, detection rates are approximately 70% and false positive rates are approximately 0.4%, even when the classifier is based only on single-packet features. The false positive drops to less 0.2% with the same 70% detection as the classifier incorporates additional features. Although this false positive rate is likely still too high for *SNARE* to subsume all other spam filtering techniques, we believe that the performance may be good enough to be used in conjunction with other methods, perhaps as an early-stage classifier, or as a substitute for conventional IP reputation systems (e.g., SpamHaus).

4.2 Accuracy of Reputation Engine

In this section, we evaluate *SNARE*’s accuracy on three different groups of features. Surprisingly, we find that, even relying on only single-packet features, *SNARE* can automatically distinguish spammers from legitimate senders. Adding additional features based on single-header or single-message, or aggregates of these features based on 24 hours of history, improves the accuracy further.

(a) Single Packet			(b) Single Header/Message			(c) 24+ Hour History		
	Classified as			Classified as			Classified as	
	Spam	Ham		Spam	Ham		Spam	Ham
Spam	70%	30%	Spam	70%	30%	Spam	70%	30%
Ham	0.44%	99.56%	Ham	0.29%	99.71%	Ham	0.20%	99.80%

Table 1: *SNARE* performance using *RuleFit* on different sets of features using covariant shift. Detection and false positive rates are shown in bold. (The detection is fixed at 70% for comparison, in accordance with today’s DNSBLs [10]).

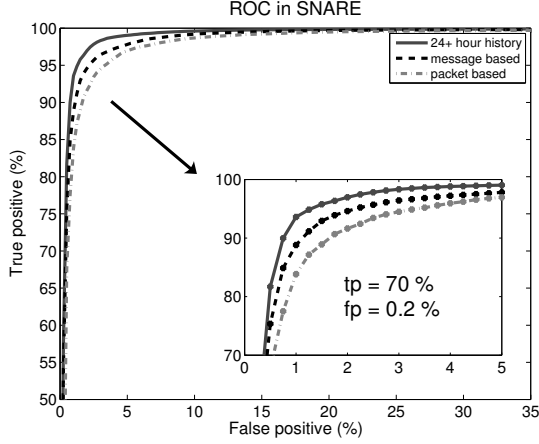


Figure 8: ROC in *SNARE*.

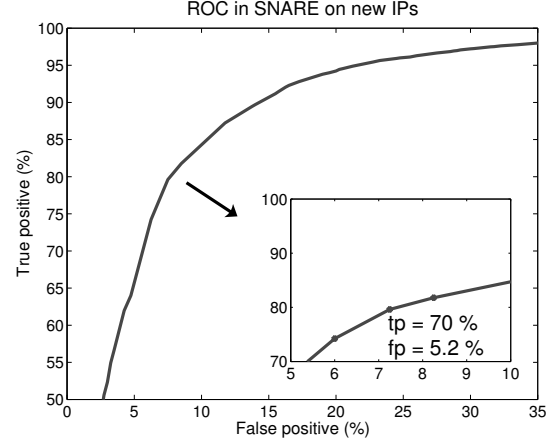


Figure 9: ROC on fresh IPs in *SNARE*.

4.2.1 Single-Packet Features

When a mail server receives a new connection request, the server can provide *SNARE* with the IP addresses of the sender and the recipient and the time-stamp based on the TCP SYN packet alone. Recall from Section 3 even if *SNARE* has never seen this IP address before, it can still combine this information with recent history of behavior of other email servers and construct the following features: (1) geodesic distance between the sender and the recipient, (2) average distance to the 20 nearest neighbors of the sender in the log, (3) probability ratio of spam to ham at the time the connection is requested (4) AS number of the sender’s IP, and (5) status of the email-service ports on the sender.

To evaluate the effectiveness of these features, we trained *RuleFit* on these features. The dash-dot curve in Figure 8 demonstrate the ROC curve of *SNARE*’s reputation engine. The fp = 0.2% and tp = 70% statistics refer to the curve with 24-hour history (solid line), which will be addresses later. We check the false positive given a fixed true positive, 70%. The confusion matrix is shown in Table 1(a). Just over 0.44% of legitimate email gets labelled as spam. This result is significant because it relies on features constructed from a limited amount of data

and just a single IP packet from the candidate IP. Sender reputation system will be deployed in conjunction with a combination of other techniques including content based filtering. As such, as a first line of defense, this system will be very effective in eliminating a lot of undesired senders. In fact, once a sender is identified as a spammer, the mail server does not even need to accept the connection request, saving network bandwidth and computational resources. The features we describe below improve accuracy further.

4.2.2 Single-Header and Single-Message Features

Single-packet features allow *SNARE* to rapidly identify and drop connections from spammers even before looking at the message header. Once a mail server has accepted the connection and examined the entire message, *SNARE* can determine sender reputation with increased confidence by looking at an additional set of features. As described in Section 3.2, these features include the number of recipients and message body length. Table 1(b) shows the prediction accuracy when we combine the single-packet features (i.e., those from the previous section) with these additional features. As the results from Section 3 suggest, adding the *message body length* and

number of recipients to the set of features further improves *SNARE*'s detection rate and false positive rate.

It is worth mentioning that the number of recipients listed on the "To" field is perhaps somewhat evadable: a sender could list the target email addresses on "Cc" and "Bcc" fields. Besides, if the spammers always place a single recipient address in the "To" field, this value will be the same as the large majority of legitimate messages. Because we did not have logs of additional fields in the SMTP header beyond the count of email addresses on the "To" field, we could not evaluate whether considering number of recipients listed under "Cc" and "Bcc" headers is worthwhile.

4.2.3 Aggregate Features

If multiple messages from a sender are available, the following features can be computed: the mean and variance of geodesic distances, message body lengths and number of recipients. We evaluate a classifier that is trained on *aggregate statistics* from the past 24 hours together with the features from previous sections.

Table 1(c) shows the performance of *RuleFit* with these aggregate features, and the ROC curve is plotted as the solid one in Figure 8. Applying the aggregate features decreases the error rate further: 70% of spam is identified correctly, while the false positive rate is merely 0.20%. The content-based filtering is very efficient to identify spam, but can not satisfy the requirement of processing a huge amount of messages for big mail servers. The prediction phase of *RuleFit* is faster, where the query is traversed from the root of the decision tree to a bottom label. Given the low false positive rate, *SNARE* would be a perfect first line of defense, where suspicious messages are dropped or re-routed to a farm for further analysis.

4.3 Other Considerations

Detection of "fresh" spammers We examined data trace, extracted the IP addresses not showing up in the previous training window, and further investigated the detection accuracy for those 'fresh' spammers with all *SNARE*'s features. If fixing the true positive as 70%, the false positive will increase to 5.2%, as shown in Figure 9. Compared with Figure 8, the decision on the new legitimate users becomes worse, but most of the new spammers can still be identified, which validates that *SNARE* is capable of *automatically* classifying "fresh" spammers.

Relative importance of individual features We use the fact that *RuleFit* can evaluate the *relative importance* of the features we have examined in Sections 3. Table 2 ranks all spatio-temporal features (with the most important feature at top). The top three features—AS

rank	Feature Description
1	AS number of the sender's IP
2	average of message length in previous 24 hours
3	average distance to the 20 nearest IP neighbors of the sender in the log
4	standard deviation of message length in previous 24 hours
5	status of email-service ports on the sender
6	geodesic distance between the sender and the recipient
7	number of recipient
8	average geodesic distance in previous 24 hours
9	average recipient number in previous 24 hours
10	probability ratio of spam to ham when getting the message
11	standard deviation of recipient number in previous 24 hours
12	length of message body
13	standard deviation of geodesic distance in previous 24 hours

Table 2: Ranking of feature importance in *SNARE*.

num, *avg length* and *neig density*—play an important role in separating out spammers from good senders. This result is quite promising, since most of these features are lightweight: Better yet, two of these three can be computed having received only a single packet from the sender. As we will discuss in Section 6, they are also relatively resistant to evasion.

Correlation analysis among features We use mutual information to investigate how tightly the features are coupled, and to what extent they might contain redundant information. Given two random variables, mutual information measures how much uncertainty of one variable is reduced after knowing the other (i.e., the information they share). For discrete variables, the mutual information of X and Y is calculated as: $I(X, Y) = \sum_{x,y} p(x, y) \log(\frac{p(x,y)}{p(x)p(y)})$. When logarithm base-two is used, the quantity reflects how many bits can be removed to encode one variable given the other one. Table 3 shows the mutual information between pairs of features for one day of training data (October 23, 2007). We do not show statistics from other days, but features on those days reflect similar quantities for mutual information. The features with continuous values (e.g., geodesic distance between the sender and the recipient) are transformed into discrete variables by dividing the value range into 4,000 bins (which yields good discrete approximation); we calculate mutual information over the discrete probabilities. The indexes of the features in the table are the same as the ranks in Table 2; the packet-based features are marked with black circles. We also calculate the entropy of every feature and show them next to the indices in Table 3.

The interpretation of mutual information is consistent only within a single column or row, since comparison of mutual information without any common variable is meaningless. The table, of course, begs additional analysis but shows some interesting observations. The top-ranked feature, AS number, shares high mutual information (shown in bold) with several other features, especially with feature 6, geodesic distance between sender and recipient. The aggregate features of first-order statis-

	1 (8.68)	2 (7.29)	3 (2.42)	4 (6.92)	5 (1.20)	6 (10.5)	7 (0.46)	8 (9.29)	9 (2.98)	10 (4.45)	11 (3.00)	12 (6.20)
2 (7.29)	4.04											
3 (2.42)	1.64	1.18										
4 (6.92)	3.87	4.79	1.23									
5 (1.20)	0.65	0.40	0.11	0.43								
6 (10.5)	5.20	3.42	0.88	3.20	0.35							
7 (0.46)	0.11	0.08	0.02	0.08	0.004	0.15						
8 (9.29)	5.27	5.06	1.20	4.79	0.46	5.16	0.13					
9 (2.98)	1.54	1.95	0.53	2.03	0.09	1.17	0.10	2.08				
10 (4.45)	0.66	0.46	0.07	0.49	0.02	0.87	0.006	0.85	0.13			
11 (3.00)	1.87	1.87	0.75	2.04	0.16	1.55	0.09	2.06	1.87	0.20		
12 (6.20)	2.34	2.53	0.49	2.12	0.20	2.34	0.07	2.30	0.52	0.31	0.73	
13 (8.89)	4.84	4.78	1.15	4.69	0.41	4.77	0.11	6.47	1.98	0.69	2.04	2.13

Table 3: Mutual information among features in *SNARE*; packet-based features are shown with numbers in dark circles. (The indices are the feature ranking in Table 2.)

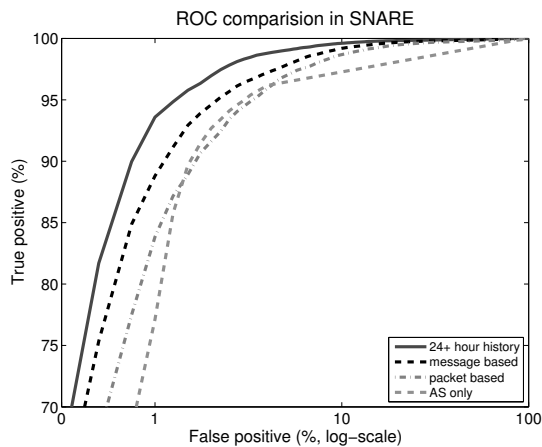


Figure 10: ROC comparison with AS-only case.

tics (e.g., feature 2, 4, 8) also have high values with each other. Because spammers may exhibit one or more of these features across each message, aggregating the features across multiple message over time indicates that, observing a spammer over time will reveal many of these features, though not necessarily on any message or single group of message. For this reason, aggregate features are likely to share high mutual information with other features that are common to spammers.

One possible reason that aggregate features have high mutual information with each other is that aggregating the features across multiple messages over time incorporates history of an IP address that may exhibit many of these characteristics over time.

Performance based on AS number only Since AS number is the most influential feature according to *Rule-Fit* and shares high mutual information with many other features, we investigated how well this feature alone can distinguish spammers from legitimate senders. We feed the AS feature into the predictive model and plot the ROC as the lower dashed curve in Figure 10. To make a

close comparison, the “packet-based”, “message-based”, and “history-based” ROCs (the same as those in Figure 8) are shown as well, and the false positive is displayed on a log scale. The classifier gets false positive 0.76% under a 70% detection rate. Recall from Table 1 the false positive rate with “packet-based” features is almost a half, 0.44%, and that with “history-based” features will further reduce to 0.20%, which demonstrates that other features help to improve the performance. We also note that using the AS number alone as a distinguishing feature may cause large amounts of legitimate email to be misclassified, and could be evaded if an spammer decides to announce routes with a forged origin AS (which is an easy attack to mount and a somewhat common occurrence) [2, 26, 39].

5 A Spam-Filtering System

This section describes how *SNARE*’s reputation engine could be integrated into an overall spam-filtering system that includes a whitelist and an opportunity to continually retrain the classifier on labeled data (e.g., from spam traps, user inboxes, etc.). Because *SNARE*’s reputation engine still has a non-zero false positive rate, we show how it might be incorporated with mechanisms that could help further improve its accuracy, and also prevent discarding legitimate mail even in the case of some false positives. We propose an overview of the system and evaluate the benefits of these two functions on overall system accuracy.

5.1 System Overview

Figure 11 shows the overall system framework. The system needs not reside on a single server. Large public email providers might run their own instance of *SNARE*, since they have plenty of email data and processing resources. Smaller mail servers might query a remote

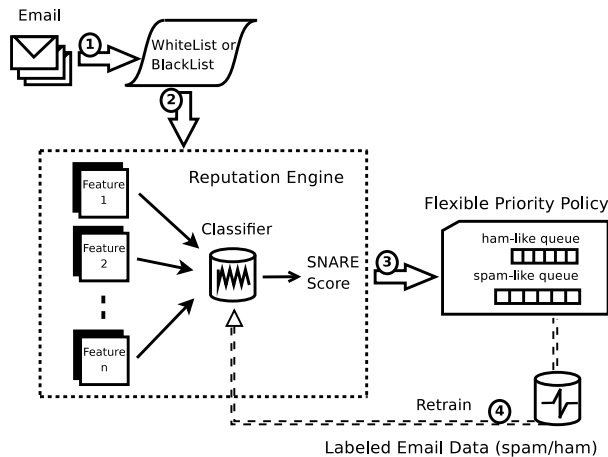


Figure 11: *SNARE* framework.

SNARE server. We envision that *SNARE* might be integrated into the workflow in the following way:

1. **Email arrival.** After getting the first packet, the mail server submits a query to the *SNARE* server (only the source and destination IP). Mail servers can choose to send more information to *SNARE* after getting the SMTP header or the whole message. Sending queries on a single packet or on a message is a tradeoff between detection accuracy and processing time for the email (i.e., sending the request early will make mail server get the response early). The statistics of messages in the received queries will be used to build up the *SNARE* classifier.
2. **Whitelisting.** The queries not listed in the whitelist will be passed to *SNARE*'s reputation engine (presented in Section 2.3) *before* any spam-filtering checks or content-based analysis. The output is a score, where, by default, positive value means likely spam and negative value means likely ham; and the absolute values represent the confidence of the classification. Administrators can set a different score threshold to make tradeoff between the false positive and the detection rate. We evaluate the benefits of whitelisting in Section 5.2.1.
3. **Greylisting and content-based detection.** Once the reputation engine calculates a score, the email will be delivered into different queues. More resource-sensitive and time-consuming detection methods (e.g., content-based detection) can be applied at this point. When the mail server has the capability to receive email, the messages in ham-like queue have higher priority to be processed, whereas the messages in spam-like queue will be offered less resources. This policy allows the server to speed up

processing the messages that *SNARE* classifies as spam. The advantage of this hierarchical detecting scheme is that the legitimate email will be delivered to users' inbox sooner. Messages in the spam-like queue could be shunted to more resource-intensive spam filters before they are ultimately dropped.³

4. **Retraining** Whether the IP address sends spam or legitimate mail in that connection is not known at the time of the request, but is known after mail is processed by the spam filter. *SNARE* depends on accurately labelled training data. The email will eventually receive more careful checks (shown as "Retrain" in Figure 11). The results from those filters are considered as ground truth and can be used as feedback to dynamically adjust the *SNARE* threshold. For example, when the mail server has spare resource or much email in the spam-like queue is considered as legitimate later, *SNARE* system will be asked to act more generous to score email as likely ham; on the other hand, if the mail server is overwhelmed or the ham-like queue has too many incorrect labels, *SNARE* will be less likely to put email into ham-like queue. Section 5.2.2 evaluates the benefits of retraining for different intervals.

5.2 Evaluation

In this section, we evaluate how the two additional functions (whitelisting and retraining) improve *SNARE*'s overall accuracy.

5.2.1 Benefits of Whitelisting

We believe that a whitelist can help reduce *SNARE*'s overall false positive rate. To evaluate the effects of such a whitelist, we examined the features associated with the false positives, and determine that, 43% of all of *SNARE*'s false positives for a single day originate from just 10 ASes. We examined this characteristic for different days and found that 30% to 40% of false positives from any given day originate from the top 10 ASes. Unfortunately, however, these top 10 ASes do not remain the same from day-to-day, so the whitelist may need to be retrained periodically. It may also be the case that other features besides AS number of the source provide an even better opportunity for whitelisting. We leave the details of refining the whitelist for future work.

Figure 12 shows the average ROC curve when we whitelist the top 50 ASes responsible for most misclassified ham in each day. This whitelisting reduces the best

³Although *SNARE*'s false positive rates are quite low, some operators may feel that any non-zero chance that legitimate mail or sender might be misclassified warrants at least a second-pass through a more rigorous filter.

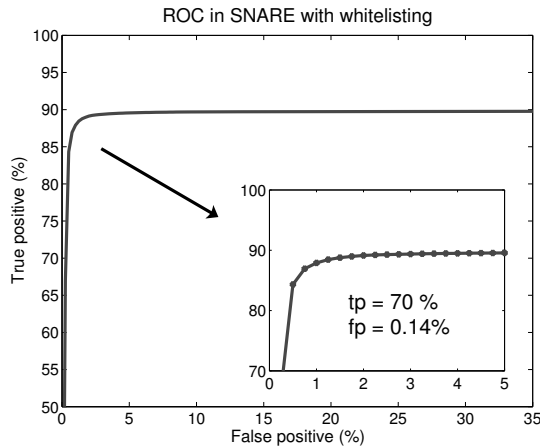


Figure 12: ROC in *SNARE* with whitelisting on ASes.

possible detection rate considerably (effectively because about 11% of spam originates from those ASes). However, this whitelisting also reduces the false positive rate to about 0.14% for a 70% detection rate. More aggressive whitelisting, or whitelisting of other features, could result in even lower false positives.

5.2.2 Benefits of Retraining

Setup Because email sender behavior is dynamic, training *SNARE* on data from an earlier time period may eventually grow stale. To examine the requirements for periodically retraining the classifier, we train *SNARE* based on the first 3 days' data (through October 23–25, 2007) and test on the following 10 days. As before, we use 1 million randomly sampled spam and ham messages to test the classifier for each day.

Results Figure 13 shows the false positive and true positive on 3 future days, October 26, October 31, and November 4, 2007, respectively. The prediction on future days will become more inaccurate with time passage. For example, on November 4 (ten days after training), the false positive rate has dropped given the same true positive on the ROC curve. This result suggests that, for the spammer behavior in this trace, retraining *SNARE*'s classification algorithms daily should be sufficient to maintain accuracy. (We expect that the need to retrain may vary across different datasets.)

6 Discussion and Limitations

In this section, we address various aspects of *SNARE* that may present practical concerns. We first discuss the extent to which an attacker might be able to evade various features, as well as the extent to which these

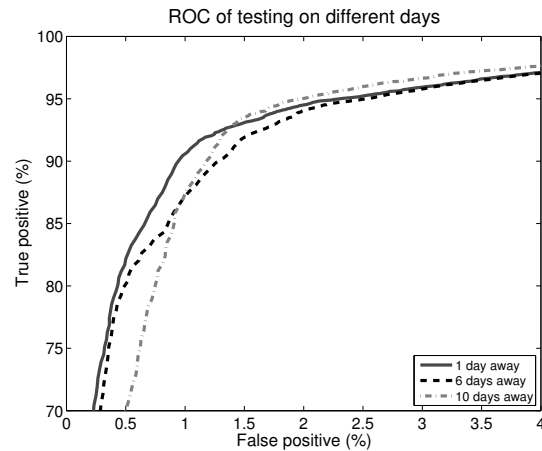


Figure 13: ROC using previous training rules to classify future messages.

features might vary across time and datasets. We then discuss scalability concerns that a production deployment of *SNARE* may present, as well as various possible workarounds.

6.1 Evasion-Resistance and Robustness

In this section, we discuss the evasion resistance of the various network-level features that form the inputs to *SNARE*'s classification algorithm. Each of these features is, to some degree, evadable. Nevertheless, *SNARE* raises the bar by making it more difficult for spammers to evade detection without altering the techniques that they use to send spam. Although spammers might adapt to evade some of the features below, we believe that it will be difficult for a spammer to adjust all features to pass through *SNARE*, particularly without somewhat reducing the effectiveness of the spamming botnet. We survey each of the features from Table 2 in turn.

AS number AS numbers are more persistently associated with a sender's identity than the IP address, for two reasons: (1) The spamming mail server might be set up within specific ASes without the network administrator shutting it down. (2) Bots tend to aggregate within ASes, since the machines in the same ASes are likely to have the same vulnerability. It is not easy for spammers to move mail servers or the bot armies to a different AS; therefore, AS numbers are robust to indicate malicious hosts.

Message length In our analysis, we discovered that the size of legitimate email messages tends to be much more variable than that of spam (perhaps because spammers often use templates to sent out large quantities of mail [25]). With knowledge of this feature, a spammer might start to randomize the lengths of their email mes-

sages; this attack would not be difficult to mount, but it might restrict the types of messages that a spammer could send or make it slightly more difficult to coordinate a massive spam campaign with similar messages.

Nearest neighbor distances Nearest neighbor distance is another feature that will be hard to modify. Distances to k nearest neighbors effectively isolate existence of unusually large number of email servers within a small sequence of IP addresses. If the spammers try to alter their neighborhood density, they will not be able to use too many machines within a compromised subnet to send spam to the same set of destinations. Although it is possible for a botnet controller to direct bots on the same subnet to target different sets of destinations, such evasion does require more coordination and, in some cases, may restrict the agility that each spamming bot has in selecting its target destinations.

Status of email service ports Some limitation might fail the active probes, e.g., the outgoing mail servers use own protocol to mitigate messages (such as Google mail) or a firewall blocks the connections from out of the domain. But the bots do not open such ports with high probability, and the attackers need to get root privilege to enable those ports (which requires more sophisticated methods and resources). The basic idea is to find out whether the sender is a legitimate mail server. Although we used active probes in *SNARE*, other methods could facilitate the test, such as domain name checking or mail server authentication.

Sender-receiver geodesic distance The distribution of geodesic distances between the spammers' physical location and their target IP's location is a result of the spammers' requirement to reach as many target mail boxes as possible and in the shortest possible time. Even in a large, geographically distributed botnet, requiring each bot to bias recipient domains to evade this feature may limit the flexibility of how the botnet is used to send spam. Although this feature can also be evaded by tuning the recipient domains for each bot, if bots only sent spam to nearby recipients, the flexibility of the botnet is also somewhat restricted: it would be impossible, for example, to mount a coordinate spam campaign against a particular region from a fully distributed spamming botnet.

Number of recipients We found that spam messages tend to have more recipients than legitimate messages; a spammer could likely evade this feature by reducing the number of recipients on each message, but this might make sending the messages less efficient, and it might alter the sender behavior in other ways that might make a spammer more conspicuous (e.g., forcing the spammer

to open up more connections).

Time of day This feature may be less resistant to evasion than others. Having said that, spamming botnets' diurnal pattern results from when the infected machines are switched on. For botnets to modify their diurnal message volumes over the day to match the legitimate message patterns, they will have to lower their spam volume in the evenings, especially between 3:00 p.m. and 9:00 p.m. and also reduce email volumes in the afternoon. This will again reduce the ability of botnets to send large amounts of email.

6.2 Other Limitations

We briefly discuss other current limitations of *SNARE*, including its ability to scale to a large number of recipients and its ability to classify IP addresses that send both spam and legitimate mail.

Scale *SNARE* must ultimately scale to thousands of domains and process hundreds of millions of email addresses per day. Unfortunately, even state-of-the-art machine learning algorithms are not well equipped to process datasets this large; additionally, sending data to a central coordinator for training could potentially consume considerably bandwidth. Although our evaluation suggests that *SNARE*'s classification is relatively robust to sampling of training data, we intend to study further the best ways to sample the training data, or perhaps even perform in-network classification.

Dual-purpose IP addresses Our conversations with large mail providers suggest that one of the biggest emerging threats are "web bots" that send spam from Web-based email accounts [35]. As these types of attacks develop, an increasing fraction of spam may be sent from IP addresses that also send significant amounts of legitimate mail. These cases, where an IP address is neither good nor bad, will need more sophisticated classifiers and features, perhaps involving timeseries-based features.

7 Related Work

We survey previous work on characterizing the network-level properties and behavior of email senders, email sender reputation systems, and other email filtering systems that are not based on content.

Characterization studies Recent characterization studies have provided increasing evidence that spammers have distinct network-level behavioral patterns. Ramachandran *et al.* [34] showed that spammers utilize transient botnets to spam at low rate from any specific IP to any domain. Xie *et al.* [38] discovered that a vast

majority of mail servers running on dynamic IP address were used solely to send spam. In their recently published study [37], they demonstrate a technique to identify bots by using signatures constructed from URLs in spam messages. Unlike *SNARE*, their signature-based botnet identification differs heavily on analyzing message content. Others have also examined correlated behavior of botnets, primarily for characterization as opposed to detection [25, 31]. Pathak *et al.* [29] deployed a relay sinkhole to gather data from multiple spam senders destined for multiple domains. They used this data to demonstrate how spammers utilize compromised relay servers to evade detection; this study looked at spammers from multiple vantage points, but focused mostly on characterizing spammers rather than developing new detection mechanisms. Niu *et al.* analyzed network-level behavior of Web spammers (e.g., URL redirections and “doorway” pages) and proposed using context-based analysis to defend against Web spam [28].

Sender reputation based on network-level behavior SpamTracker [34] is most closely related to *SNARE*; it uses network-level behavioral features from data aggregated across multiple domains to infer sender reputation. While that work initiated the idea of *behavioral blacklisting*, we have discovered many other features that are more lightweight and more evasion-resistant than the single feature used in that paper. Beverly and Sollins built a similar classifier based on transport-level characteristics (e.g., round-trip times, congestion windows) [12], but their classifier is both heavyweight, as it relies on SVM, and it also requires accepting the messages to gather the features. Tang *et al.* explored the detection of spam senders by analyzing the behavior of IP addresses as observed by query patterns [36]. Their work focuses on the breadth and the periodicity of message volumes in relation to sources of queries. Various previous work has also attempted to cluster email senders according to groups of recipients, often with an eye towards spam filtering [21, 24, 27], which is similar in spirit to *SNARE*’s geodesic distance feature; however, these previous techniques typically require analysis of message contents, across a large number of recipients, or both, whereas *SNARE* can operate on more lightweight features. McAfee’s TrustedSource [4] and Cisco IronPort [3] deploy spam filtering appliances to hundreds or thousands of domains which then query the central server for sender reputation and also provide meta-data about messages they receive; we are working with McAfee to deploy *SNARE* as part of TrustedSource.

Non-content spam filtering Trinity [14] is a distributed, content-free spam detection system for messages originating from botnets that relies on message volumes. The SpamHINTS project [9] also has the stated goal of build-

ing a spam filter using analysis of network traffic patterns instead of the message content. Clayton’s earlier work on extrusion detection involves monitoring of server logs at both the local ISP [16] as well as the remote ISP [17] to detect spammers. This work has similar objectives as ours, but the proposed methods focus more on properties related to SMTP sessions from only a single sender.

8 Conclusion

Although there has been much progress in content-based spam filtering, state-of-the-art systems for *sender reputation* (e.g., DNSBLs) are relatively unresponsive, incomplete, and coarse-grained. Towards improving this state of affairs, this paper has presented *SNARE*, a sender reputation system that can accurately and automatically classify email senders based on features that can be determined early in a sender’s history—sometimes after seeing only a single IP packet.

Several areas of future work remain. Perhaps the most uncharted territory is that of using temporal features to improve accuracy. All of *SNARE*’s features are essentially discrete variables, but we know from experience that spammers and legitimate senders also exhibit different temporal patterns. In a future version of *SNARE*, we aim to incorporate such temporal features into the classification engine. Another area for improvement is making *SNARE* more evasion-resistant. Although we believe that it will be difficult for a spammer to evade *SNARE*’s features and still remain effective, designing classifiers that are more robust in the face of active attempts to evade and mis-train the classifier may be a promising area for future work.

Acknowledgments

We thank our shepherd, Vern Paxson, for many helpful suggestions, including the suggestions to look at mutual information between features and several other improvements to the analysis and presentation. We also thank Wenke Lee, Anirudh Ramachandran, and Mukarram bin Tariq for helpful comments on the paper. This work was funded by NSF CAREER Award CNS-0643974 and NSF Awards CNS-0716278 and CNS-0721581.

References

- [1] FCrDNS Lookup Testing. <http://ipadmin.junkemailfilter.com/rdns.php>.
- [2] Internet Alert Registry. <http://iar.cs.unm.edu/>.
- [3] IronPort. <http://www.ironport.com>.
- [4] McAfee Secure Computing. <http://www.securecomputing.com>.
- [5] SORBS: Spam and Open Relay Blocking System. <http://www.au.sorbs.net/>.
- [6] SpamCop. <http://www.spamcop.net/bl.shtml>.
- [7] SpamHaus IP Blocklist. <http://www.spamhaus.org>.
- [8] GeoIP API. MaxMind, LLC. <http://www.maxmind.com/app/api>, 2007.
- [9] spamHINTS: Happily It's Not The Same. <http://www.spamhints.org/>, 2007.
- [10] DNSBL Resource: Statistics Center. <http://stats.dnsbl.com/>, 2008.
- [11] ALPEROVITCH, D., JUDGE, P., AND KRASSER, S. Taxonomy of email reputation systems. In *Proc. of the First International Workshop on Trust and Reputation Management in Massively Distributed Computing Systems (TRAM)* (2007).
- [12] BEVERLY, R., AND SOLLINS, K. Exploiting the transport-level characteristics of spam. In *5th Conference on Email and Anti-Spam (CEAS)* (2008).
- [13] BOYKIN, P., AND ROYCHOWDHURY, V. Personal email networks: An effective anti-spam tool. *IEEE Computer* 38, 4 (2005), 61–68.
- [14] BRODSKY, A., AND BRODSKY, D. A distributed content independent method for spam detection. In *First Workshop on Hot Topics in Understanding Botnets (HotBots)* (2007).
- [15] BURGESS, C. A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery* 2, 2 (1998), 121–167.
- [16] CLAYTON, R. Stopping spam by extrusion detection. In *First Conference of Email and Anti-Spam (CEAS)* (2004).
- [17] CLAYTON, R. Stopping outgoing spam by examining incoming server logs. In *Second Conference on Email and Anti-Spam (CEAS)* (2005).
- [18] FRIEDMAN, J., AND POPESCU, B. Gradient directed regularization. *Stanford University, Technical Report* (2003).
- [19] FRIEDMAN, J., AND POPESCU, B. Predictive learning via rule ensembles. *Annals of Applied Statistics (to appear)* (2008).
- [20] GOLBECK, J., AND HENDLER, J. Reputation network analysis for email filtering. In *First Conference on Email and Anti-Spam (CEAS)* (2004).
- [21] GOMES, L. H., CASTRO, F. D. O., ALMEIDA, R. B., BETENCOURT, L. M. A., ALMEIDA, V. A. F., AND ALMEIDA, J. M. Improving spam detection based on structural similarity. In *Proceedings of the Steps to Reducing Unwanted Traffic on the Internet Workshop (SRUTI)* (2005).
- [22] GOODMAN, J., CORMACK, G., AND HECKERMAN, D. Spam and the ongoing battle for the inbox. *Communications of the ACM* 50, 2 (2007), 24–33.
- [23] HULTON, E., AND GOODMAN, J. Tutorial on junk email filtering. *Tutorial in the 21st International Conference on Machine Learning (ICML)* (2004).
- [24] JOHANSEN, L., ROWELL, M., BUTLER, K., AND MCDANIEL, P. Email communities of interest. In *4th Conference on Email and Anti-Spam (CEAS)* (2007).
- [25] KANICH, C., KREIBICH, C., LEVCHENKO, K., ENRIGHT, B., PAXSON, V., VOELKER, G. M., AND SAVAGE, S. Spamalytics: An empirical analysis of spam marketing conversion. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)* (2008).
- [26] KARLIN, J., FORREST, S., AND REXFORD, J. Autonomous security for autonomous systems. *Computer Networks* 52, 15 (2008), 2908–2923.
- [27] LAM, H., AND YEUNG, D. A learning approach to spam detection based on social networks. In *4th Conference on Email and Anti-Spam (CEAS)* (2007).
- [28] NIU, Y., WANG, Y.-M., CHEN, H., MA, M., AND HSU, F. A quantitative study of forum spamming using context-based analysis. In *Proceedings of the 14th Annual Network and Distributed System Security Symposium (NDSS)* (2007).
- [29] PATHAK, A., HU, C., Y., AND MAO, Z., M. Peeking into spammer behavior from a unique vantage point. In *First USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)* (2008).
- [30] QUINLAN, J. Induction of decision trees. *Machine Learning* 1, 1 (1986), 81–106.
- [31] RAJAB, M., ZARFOSS, J., MONROSE, F., AND TERZIS, A. A multifaceted approach to understanding the botnet phenomenon. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement (IMC)* (2006).
- [32] RAMACHANDRAN, A., DAGON, D., AND FEAMSTER, N. Can DNSBLs keep up with bots? In *3rd Conference on Email and Anti-Spam (CEAS)* (2006).
- [33] RAMACHANDRAN, A., AND FEAMSTER, N. Understanding the network-level behavior of spammers. In *Proceedings of the ACM SIGCOMM* (2006).
- [34] RAMACHANDRAN, A., FEAMSTER, N., AND VEMPALA, S. Filtering spam with behavioral blacklisting. In *ACM Conference on Computer and Communications Security (CCS)* (2007).
- [35] Private conversation with Mark Risher, Yahoo Mail., 2008.
- [36] TANG, Y. C., KRASSER, S., JUDGE, P., AND ZHANG, Y.-Q. Fast and effective spam IP detection with granular SVM for spam filtering on highly imbalanced spectral mail server behavior data. In *2nd International Conference on Collaborative Computing (CollaborateCom)* (2006).
- [37] XIE, Y., YU, F., , ACHAN, K., PANIGRAHY, R., HULTEN, G., AND OSIPKOV, I. Spamming bots: Signatures and characteristics. In *Proceedings of ACM SIGCOMM* (2008).
- [38] XIE, Y., YU, F., ACHAN, K., GILUM, E., GOLDSZMIDT, M., AND WOBBER, T. How dynamic are IP addresses. In *Proceedings of ACM SIGCOMM* (2007).
- [39] ZHAO, X., PEI, D., WANG, L., MASSEY, D., MANKIN, A., WU, S. F., AND ZHANG, L. An analysis of BGP multiple origin AS (MOAS) conflicts. In *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement (IMW)* (2001).

Improving Tor using a TCP-over-DTLS Tunnel

Joel Reardon*
Google Switzerland GmbH
Brandschenkestrasse 110
Zürich, Switzerland
reardon@google.com

Ian Goldberg
University of Waterloo
200 University Ave W.
Waterloo, ON, Canada
iang@cs.uwaterloo.ca

Abstract

The Tor network gives anonymity to Internet users by relaying their traffic through the world over a variety of routers. All traffic between any pair of routers, even if they represent circuits for different clients, are multiplexed over a single TCP connection. This results in interference across circuits during congestion control, packet dropping and packet reordering. This interference greatly contributes to Tor's notorious latency problems.

Our solution is to use a TCP-over-DTLS (Datagram Transport Layer Security) transport between routers. We give each stream of data its own TCP connection, and protect the TCP headers—which would otherwise give stream identification information to an attacker—with DTLS. We perform experiments on our implemented version to illustrate that our proposal has indeed resolved the cross-circuit interference.

1 Introduction

Tor [2] is a tool to enable Internet privacy that has seen widespread use and popularity throughout the world. Tor consists of a network of thousands of nodes—known as Onion Routers (ORs)—whose operators have volunteered to relay Internet traffic around the world. Clients—known as Onion Proxies (OPs)—build circuits through ORs in the network to dispatch their traffic. Tor's goal is to frustrate an attacker who aims to match up the identities of the clients with the actions they are performing. Despite its popularity, Tor has a problem that dissuades its ubiquitous application—it imposes greater latency on its users than they would experience without Tor.

While some increased latency is inevitable due to the increased network path length, our experiments show that this effect is not sufficient to explain the increased cost. In Section 2 we look deeper, and find a component

of the transport layer that can be changed to improve Tor's performance. Specifically, each pair of routers maintains a single TCP connection for all traffic that is sent between them. This includes multiplexed traffic for different circuits, and results in cross-circuit interference that degrades performance. We find that congestion control mechanisms are being unfairly applied to all circuits when they are intended to throttle only the noisy senders. We also show how packet dropping on one circuit causes interference on other circuits.

Section 3 presents our solution to this problem—a new transport layer that is backwards compatible with the existing Tor network. Routers in Tor can gradually and independently upgrade, and our system provides immediate benefit to any pair of routers that choose to use our improvements. It uses a separate TCP connection for each circuit, but secures the TCP header to avoid the disclosure of per-circuit data transfer statistics. Moreover, it uses a user-level TCP implementation to address the issue of socket proliferation that prevents some operating systems from being able to volunteer as ORs.

Section 4 presents experiments to compare the existing Tor with our new implementation. We compare latency and throughput, and perform timing analysis of our changes to ensure that they do not incur non-negligible computational latency. Our results are favourable: the computational overhead remains negligible and our solution is successful in addressing the improper use of congestion control.

Section 5 compares our enhanced Tor to other anonymity systems, and Section 6 concludes with a description of future work.

1.1 Apparatus

Our experiments were performed on a commodity Thinkpad R60—1.66 GHz dual core with 1 GB of RAM. Care was taken during experimentation to ensure that the system was never under load significant enough to influ-

*Work done while at the University of Waterloo

ence the results. Our experiments used a modified version of the Tor 0.2.0.x stable branch code.

2 Problems with Tor's Transport Layer

We begin by briefly describing the important aspects of Tor's current transport layer. For more details, see [2]. An end user of Tor runs an Onion Proxy on her machine, which presents a SOCKS proxy interface [7] to local applications, such as web browsers. When an application makes a TCP connection to the OP, the OP splits it into fixed-size *cells* which are encrypted and forwarded over a *circuit* composed of (usually 3) Onion Routers. The last OR creates a TCP connection to the intended destination host, and passes the data between the host and the circuit.

The circuit is constructed with hop-by-hop TCP connections, each protected with TLS [1], which provides confidentiality and data integrity. The OP picks a first OR (OR_1), makes a TCP connection to it, and starts TLS on that connection. It then instructs OR_1 to connect to a particular second OR (OR_2) of the OP's choosing. If OR_1 and OR_2 are not already in contact, a TCP connection is established between them, again with TLS. If OR_1 and OR_2 are already in contact (because other users, for example, have chosen those ORs for their circuits), the existing TCP connection is used for all traffic between those ORs. The OP then instructs OR_2 to contact a third OR, OR_3 , and so on. Note that there is *not* an end-to-end TCP connection from the OP to the destination host, nor to any OR except OR_1 .

This multi-hop transport obviously adds additional unavoidable latency. However, the observed latency of Tor is larger than accounted for simply by the additional transport time. In [12], the first author of this paper closely examined the sources of latency in a live Tor node. He found that processing time and input buffer queueing times were negligible, but that output buffer queueing times were significant. For example, on an instrumented Tor node running on the live Tor network, 40% of output buffers had data waiting in them from 100 ms to over 1 s more than 20% of the time. The data was waiting in these buffers because the operating system's output buffer for the corresponding socket was itself full, and so the OS was reporting the socket as unwritable. This was due to TCP's congestion control mechanism, which we discuss next.

Socket output buffers contain two kinds of data: packet data that has been sent over the network but is unacknowledged¹, and packet data that has not been sent due to TCP's congestion control. Figure 1 shows the

¹Recall that TCP achieves reliability by buffering all data locally until it has been acknowledged, and uses this to generate retransmission messages when necessary

size of the socket output buffer over time for a particular connection. First, unwritable sockets occur when the remaining capacity in an output buffer is too small to accept new data. This in turn occurs because there is already too much data in the buffer, which is because there is too much unacknowledged data in flight and throttled data waiting to be sent. The congestion window (CWND) is a variable that stores the number of packets that TCP is currently willing to send to the peer. When the number of packets in flight exceeds the congestion window then the sending of more data is throttled until acknowledgments are received. Once congestion throttles sending, the data queues up until either packets are acknowledged or the buffer is full.

In addition to congestion control, TCP also has a flow control mechanism. Receivers advertise the amount of data they are willing to accept; if more data arrives at the receiver before the receiving application has a chance to read from the OS's receive buffers, this advertised receiver window will shrink, and the sender will stop transmitting when it reaches zero. In none of our experiments did we ever observe Tor throttling its transmissions due to this mechanism; the advertised receiver window sizes never dropped to zero, or indeed below 50 KB. Congestion control, rather than flow control, was the reason for the throttling.

While data is delayed because of congestion control, it is foolhardy to attempt to circumvent congestion control as a means of improving Tor's latency. However, we observe that Tor's transport between ORs results in an *unfair* application of congestion control. In particular, Tor's circuits are multiplexed over TCP connections; i.e., a single TCP connection between two ORs is used for multiple circuits. When a circuit is built through a pair of unconnected routers, a new TCP connection is established. When a circuit is built through an already-connected pair of ORs, the existing TCP stream will carry both the existing circuits and the new circuit. This is true for all circuits built in either direction between the ORs.

In this section we explore how congestion control affects multiplexed circuits and how packet dropping and reordering can cause interference across circuits. We show that TCP does not behave optimally when circuits are multiplexed in this manner.

2.1 Unfair Congestion Control

We believe that multiplexing TCP streams over a single TCP connection is unwise and results in the unfair application of TCP's congestion control mechanism. It results in multiple data streams competing to send data over a TCP stream that gives more bandwidth to circuits that send more data; i.e., it gives each byte of data the same priority regardless of its source. A busy circuit that

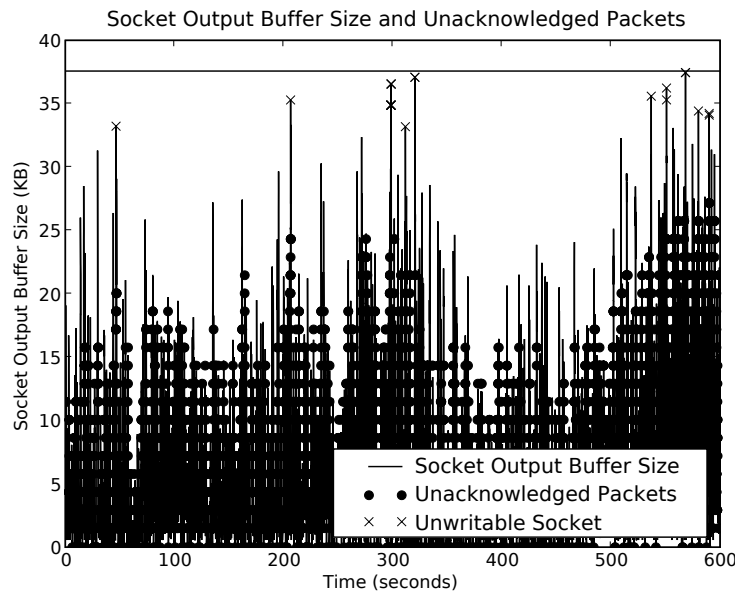


Figure 1: TCP socket output buffer size, writability, and unacknowledged packets over time.

triggers congestion control will cause low-bandwidth circuits to struggle to have their data sent. Figure 2 illustrates data transmission for distinct circuits entering and exiting a single output buffer in Tor. Time increases along the X-axis, and data increases along the Y-axis. The main part of the figure shows two increasing line shapes, each corresponding to the data along a different circuit over time. When the shapes swell, that indicates that Tor’s internal output buffer has swelled: the left edge grows when data enters the buffer, and the right edge grows when data leaves the buffer. This results in the appearance of a line when the buffer is well-functioning, and a triangular or parallelogram shape when data arrives too rapidly or the connection is troubled. Additionally, we strike a vertical line across the graph whenever a packet is dropped.

What we learn from this graph is that the buffer serves two circuits. One circuit serves one MB over ten minutes, and sends cells evenly. The other circuit is inactive for the most part, but three times over the execution it suddenly serves 200 KB of cells. We can see that each time the buffer swells with data it causes a significant delay. Importantly, the other circuit is affected despite the fact that it did not change its behaviour. Congestion control mechanisms that throttle the TCP connection will give preference to the burst of writes because it simply provides more data, while the latency for a low-bandwidth application such as `ssh` increases unfairly.

2.2 Cross-Circuit Interference

Tor multiplexes the data for a number of circuits over a single TCP stream, and this ensures that the received data will appear in the precise order in which the component streams were multiplexed—a guarantee that goes beyond what is strictly necessary. When packets are dropped or reordered, the TCP stack will buffer available data on input buffers until the missing in-order component is available. We hypothesize that when active circuits are multiplexed over a single TCP connection, Tor suffers an unreasonable performance reduction when either packet dropping or packet reordering occur. Cells may be available in-order for one particular circuit but are being delayed due to missing cells for another circuit. In-order guarantees are only necessary for data sent within a single circuit, but the network layer ensures that data is only readable in the order it was dispatched. Packet loss or reordering will cause the socket to indicate that no data is available to read even if other circuits have their sequential cells available in buffers.

Figure 3 illustrates the classic head-of-line blocking behaviour of Tor during a packet drop; cells for distinct circuits are represented by shades and a missing packet is represented with a cross. We see that the white, light grey, and black circuits have had all of their data successfully received, yet the kernel will not pass that data to the Tor application until the dropped dark grey packet is retransmitted and successfully received.

We verify our cross-circuit interference hypothesis in two parts. In this section we show that packet drops on a

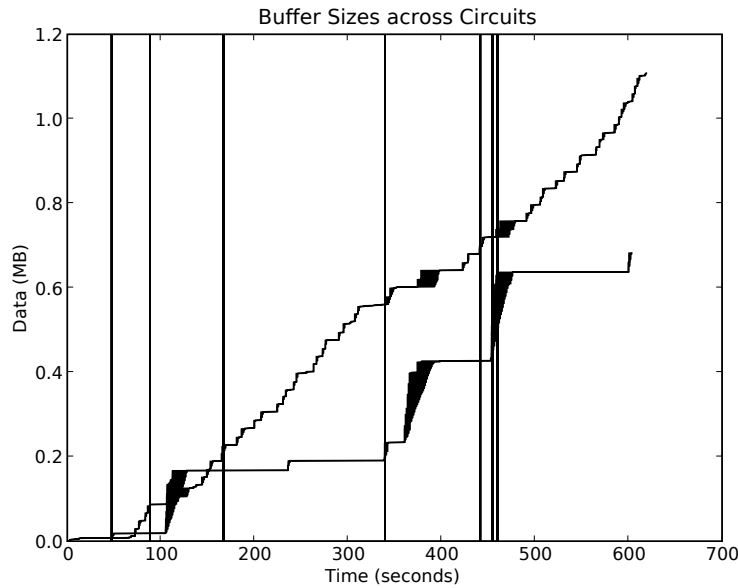


Figure 2: Example of congestion on multiple streams.

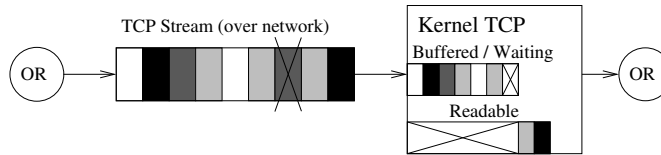


Figure 3: TCP correlated streams. Shades correspond to cells for different circuits.

Experiment 1 Determining the effect of packet dropping on circuit multiplexing.

- 1: A Tor network of six ORs on a single host was configured to have a latency of 50 milliseconds and a variable packet drop rate.
- 2: Eight OP built circuits that were fixed so that the second and third ORs were the same for each client, but the first hop was evenly distributed among the remaining ORs. Figure 4 illustrates this setup.
- 3: There were three runs of the experiment. The first did not drop any packets. The second dropped 0.1% of packets on the shared link, and the third dropped 0.1% of packets on the remaining links.
- 4: The ORs were initialized and then the clients were run until circuits were established.
- 5: Each OP had a client connect, which would tunnel a connection to a timestamp server through Tor. The server sends a continuous stream of timestamps. The volume of timestamps measures throughput, and the difference in time measures latency.
- 6: Data was collected for one minute.

shared link degrade throughput much more severely than drops over unshared links. Then in Section 4 we show that this effect disappears with our proposed solution.

To begin, we performed Experiment 1 to investigate the effect of packet dropping on circuit multiplexing. The layout of circuits in the experiment, as shown in Figure 4, is chosen so that there is one shared link that carries data for all circuits, while the remaining links do not.

In the two runs of our experiments that drop packets, they are dropped according to a target drop rate, either on the heavily shared connection or the remaining connections. Our packet dropping tool takes a packet, decides if it is eligible to be dropped in this experiment, and if so then it drops it with the appropriate probability. However, this model means the two runs that drop packets will see different rates of packet dropping systemwide, since we observe greater traffic on the remaining connections. This is foremost because it spans two hops along the circuit instead of one, and also because traffic from multiple circuits can be amalgamated into one packet for transmission along the shared connection. As a result, a fixed drop rate affecting the remaining connec-

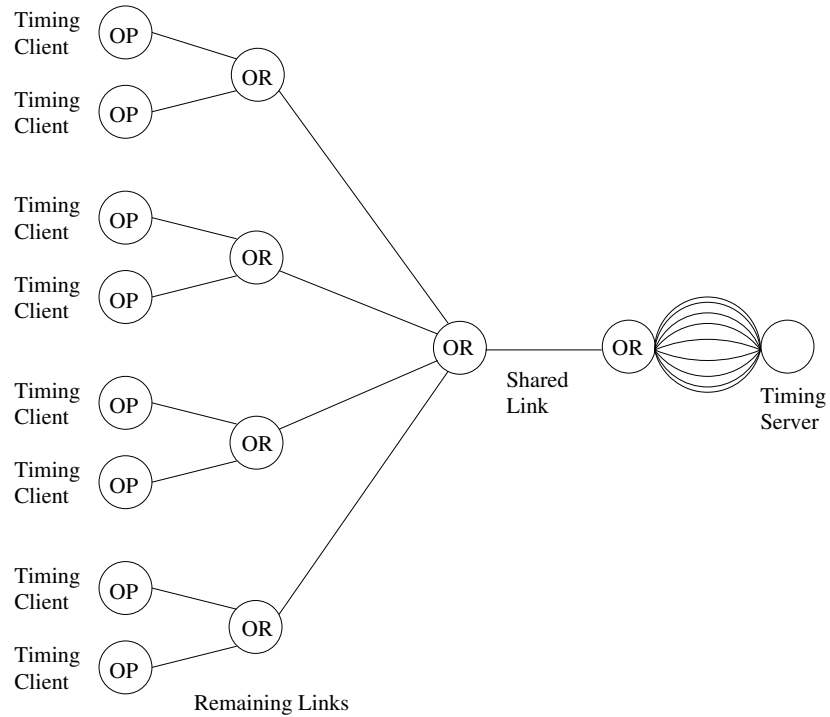


Figure 4: Setup for Experiment 1. The shared link multiplexes all circuits from the various OPs to the final OR; the remaining links carry just one or two circuits each. The splay of links between the final OR and the timing server reflect the fact that a separate TCP connection is made from the final OR to the timing server for each timing client.

Configuration	Network Throughput (KB/s)	Circuit Throughput (KB/s)	Throughput Degradation	Effective Drop Rate
No dropping	221 ± 6.6	36.9 ± 1.1	0 %	0 %
0.1 % (remaining)	208 ± 14	34.7 ± 2.3	6 %	0.08 %
0.1 % (shared)	184 ± 17	30.8 ± 2.8	17 %	0.03 %

Table 1: Throughput for different dropping configurations. Network throughput is the total data sent along all the circuits.

Configuration	Average Latency	Latency Increase	Effective Drop Rate
No dropping	933 ± 260 ms	0 %	0 %
0.1 % (remaining)	983 ± 666 ms	5.4 %	0.08 %
0.1 % (shared)	1053 ± 409 ms	12.9 %	0.03 %

Table 2: Latency for different dropping configurations.

tions will result in more frequent packet drops than one dropping only along the shared connection. This disparity is presented explicitly in our results as the effective drop rate; i.e., the ratio of packets dropped to the total number of packets we observed (including those ineligible to be dropped) in the experiment.

The results of Experiment 1 are shown in Tables 1 and 2. They show the results for three configurations: when no packet dropping is done, when 0.1% of packets are dropped on all connections except the heavily shared one, and when 0.1% of packets are dropped only on the shared connection. The degradation column refers to the loss in performance as a result of introducing packet drops. The average results for throughput and delay were accumulated over half a dozen executions of the experiment, and the mean intervals for the variates are computed using Student's T distribution to 95% confidence.

These results confirm our hypothesis. The throughput degrades nearly threefold when packets are dropped on the shared link instead of the remaining links. This is despite a significantly lower overall drop rate. The behaviour of one TCP connection can adversely affect all correlated circuits, even if those circuits are used to transport less data.

Table 2 suggests that latency increases when packet dropping occurs. Latency is measured by the time required for a single cell to travel alongside a congested circuit, and we average a few dozen such probes. Again we see that dropping on the shared link more adversely affects the observed delay despite a reduced drop rate. However, we note that the delay sees wide variance, and the 95% confidence intervals are quite large.

2.3 Summary

Multiplexing circuits over a single connection is a potential source of unnecessary latency since it causes TCP's congestion control mechanism to operate unfairly towards connections with smaller demands on throughput. High-bandwidth streams that trigger congestion control result in low-bandwidth streams having their congestion window unfairly reduced. Packet dropping and reordering also cause available data for multiplexed circuits to wait needlessly in socket buffers. These effects degrade both latency and throughput, which we have shown in experiments.

To estimate the magnitude of this effect in the real Tor network, we note that 10% of Tor routers supply 87% of the total network bandwidth [8]. A straightforward calculation shows that links between top routers—while only comprising 1% of the possible network links—transport over 75% of the data. At the time of writing, the number of OPs is estimated in the hundreds of thousands and there are only about one thousand active ORs

[14]. Therefore, even while most users are idle, the most popular 1% of links will be frequently multiplexing circuits.

Ideally, we would open a separate TCP connection for every circuit, as this would be a more appropriate use of TCP between ORs; packet drops on one circuit, for example, would not hold up packets in other circuits. However, there is a problem with this naive approach. An adversary observing the network could easily distinguish packets for each TCP connection just by looking at the port numbers, which are exposed in the TCP headers. This would allow him to determine which packets were part of which circuits, affording him greater opportunity for traffic analysis. Our solution is to tunnel packets from multiple TCP streams over DTLS, a UDP protocol that provides for the confidentiality of the traffic it transports. By tunnelling TCP over a secure protocol, we can protect both the TCP payload and the TCP headers.

3 Proposed Transport Layer

This section proposes a TCP-over-DTLS tunnelling transport layer for Tor. This tunnel transports TCP packets between peers using DTLS—a secure datagram (UDP-based) transport [9]. A user-level TCP stack running inside Tor generates and parses TCP packets that are sent over DTLS between ORs. Our solution will use a single unconnected UDP socket to communicate with all other ORs at the network level. Internally, it uses a separate user-level TCP connection for each circuit. This decorrelates circuits from TCP streams, which we have shown to be a source of unnecessary latency. The use of DTLS also provides the necessary security and confidentiality of the transported cells, including the TCP header. This prevents an observer from learning per-circuit meta-data such data how much data is being sent in each direction for individual circuits. Additionally, it reduces the number of sockets needed in kernel space, which is known to be a problem that prevents some Windows computers from volunteering as ORs. Figure 5 shows the design of our proposed transport layer, including how only a single circuit is affected by a dropped packet.

The interference that multiplexed circuits can have on each other during congestion, dropping, and reordering is a consequence of using a single TCP connection to transport data between each pair of ORs. This proposal uses a separate TCP connection for each circuit, ensuring that congestion or drops in one circuit will not affect other circuits.

3.1 A TCP-over-DTLS Tunnel

DTLS [9] is the datagram equivalent to the ubiquitous TLS protocol [1] that secures much traffic on the Inter-

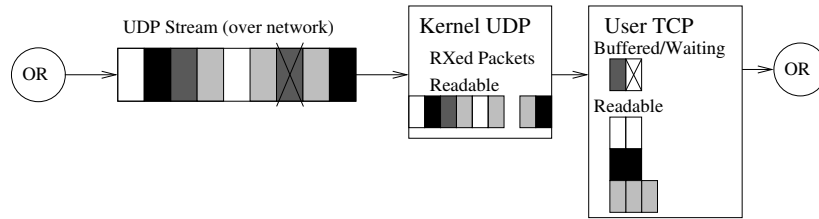


Figure 5: Proposed TCP-over-DTLS Transport showing decorrelated streams. Shades correspond to cells for different circuits (cf. Figure 3).

net today, including https web traffic, and indeed Tor. DTLS provides confidentiality and authenticity for Internet datagrams, and provides other security properties such as replay prevention. IPsec [6] would have been another possible choice of protocol to use here; however, we chose DTLS for our application due to its acceptance as a standard, its ease of use without kernel or superuser privileges, and its existing implementation in the OpenSSL library (a library already in use by Tor). The TLS and DTLS APIs in OpenSSL are also unified; after setup, the same OpenSSL calls are used to send and receive data over either TLS or DTLS. This made supporting backwards compatibility easier: the Tor code will send packets either over TCP (with TLS) or UDP (with DTLS), as appropriate, with minimal changes.

Our new transport layer employs a user-level TCP stack to generate TCP packets, which are encapsulated inside a DTLS packet that is then sent by the system in a UDP/IP datagram. The receiving system will remove the UDP/IP header when receiving data from the socket, decrypt the DTLS payload to obtain a TCP packet, and translate it into a TCP/IP packet, which is then forwarded to the user-level TCP stack that processes the packet. A subsequent read from the user-level TCP stack will provide the packet data to our system.

In our system, the TCP sockets reside in user space, and the UDP sockets reside in kernel space. The use of TCP-over-DTLS affords us the great utility of TCP: guaranteed in-order delivery and congestion control. The user-level TCP stack provides the functionality of TCP, and the kernel-level UDP stack is used simply to transmit packets. The secured DTLS transport allows us to protect the TCP header from snooping and forgery and effect a reduced number of kernel-level sockets.

ORs require opening many sockets, and so our user-level TCP stack must be able to handle many concurrent sockets, instead of relying on the operating system’s TCP implementation that varies from system to system. In particular, some discount versions of Windows artificially limit the number of sockets the user can open, and so we use Linux’s free, open-source, and high-performance TCP implementation inside user

space. Even Windows users will be able to benefit from an improved TCP implementation, and thus any user of an operating system supported by Tor will be able to volunteer their computer as an OR if they so choose.

UDP allows sockets to operate in an unconnected state. Each time a datagram is to be sent over the Internet, the destination for the packet is also provided. Only one socket is needed to send data to every OR in the Tor network. Similarly, when data is read from the socket, the sender’s address is also provided alongside the data. This allows a single socket to be used for reading from all ORs; all connections and circuits will be multiplexed over the same socket. When reading, the sender’s address can be used to demultiplex the packet to determine the appropriate connection for which it is bound. What follows is that a single UDP socket can be used to communicate with as many ORs as necessary; the number of kernel-level sockets is constant for arbitrarily many ORs with which a connection may be established. This will become especially important for scalability as the number of nodes in the Tor network grows over time. From a configuration perspective, the only new requirement is that the OR operator must ensure that a UDP port is externally accessible; since they must already ensure this for a TCP port we feel that this is a reasonable configuration demand.

Figure 6(a) shows the packet format for TCP Tor, and Figure 6(b) shows the packet format for our TCP-over-DTLS Tor, which has expanded the encrypted payload to include the TCP/IP headers generated by the user-level TCP stack. The remainder of this section will discuss how we designed, implemented, and integrated these changes into Tor.

3.2 Backwards Compatibility

Our goal is to improve Tor to allow TCP communication using UDP in the transport layer. While the original ORs transported cells between themselves, our proposal is to transport, using UDP, both TCP headers and cells between ORs. The ORs will provide the TCP/IP packets to a TCP stack that will generate both the appropriate stream of cells to the Tor application, as well

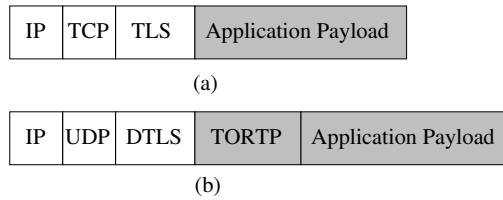


Figure 6: Packets for TCP Tor and our TCP-over-DTLS improved Tor. Encrypted and authenticated components of the packet are shaded in grey. (a) shows the packet format for TCP Tor. (b) shows the packet format for our TCP-over-DTLS Tor. TORTP is a compressed form of the IP and TCP headers, and is discussed in Section 3.4.

as TCP/IP packets containing TCP acknowledgements to be returned.

The integration of this transport layer into Tor has two main objectives. The first is that of interoperability; it is essential that the improved Tor is backwards compatible with the TCP version of Tor so as to be easily accepted into the existing codebase. Recall that Tor has thousands of ORs, a client population estimated in the hundreds of thousands, and has not experienced any downtime since it launched in 2003. It is cumbersome to arrange a synchronized update of an unknown number of anonymous Tor users. A subset of nodes that upgrade and can take advantage of TCP-over-DTLS can provide evidence of the transport’s improvement for the user experience—this incremental upgrade is our preferred path to acceptance. Our second objective is to minimize the changes required to the Tor codebase. We must add UDP connections into the existing datapath by reusing as much existing code as possible. This permits future developers to continue to improve Tor’s datapath without having to consider two classes of communication. Moreover, it encourages the changes to quickly be incorporated into the main branch of the source code. While it will come with a performance cost for doing unnecessary operations, we perform timing analyses below to ensure that the resulting datapath latency remains negligible.

Interoperability between existing ORs and those using our improved transport is achieved by fully maintaining the original TCP transport in Tor—improved ORs continue to advertise a TCP OR port and multiplexed TCP connections can continue to be made. In addition, improved nodes will also advertise a UDP port for making TCP-over-DTLS connections. Older nodes will ignore this superfluous value, while newer nodes will always choose to make a TCP-over-DTLS connection whenever such a port is advertised. Thus, two UDP nodes will automatically communicate using UDP without disrupting the existing nodes; their use of TCP-over-DTLS is inconsequential to the other nodes. As more nodes support TCP-over-DTLS, more users will obtain its benefits, but we do not require a synchronized update to support our improvements.

Clients of Tor are not required to upgrade their software to obtain the benefits of UDP transport. If two nodes on their circuit use TCP-over-DTLS to communicate then this will happen transparently to the user. In fact, it is important that the user continue to choose their circuit randomly among the ORs: intentionally choosing circuits consisting of UDP nodes when there are only a few such nodes decreases the privacy afforded to the client by rendering their circuit choices predictable.

3.3 User-level TCP Stack

If we simply replaced the TCP transport layer in Tor with a UDP transport layer, our inter-OR communication would then lack the critical features of TCP: guaranteed in-order transmission of streams, and the most well-studied congestion control mechanism ever devised. We wish to remove some of the unnecessary guarantees of TCP for the sake of latency; i.e., we do not need cells from separate circuits over the same connection to arrive in the order they were dispatched. However, we must still be able to reconstruct the streams of each individual circuit at both ends of the connection. We use a TCP implementation in user space (instead of inside the operating system) to accommodate us; a user-level TCP stack provides the implementation of the TCP protocols [4] as part of our program. User-level socket file descriptors and their associated data structures and buffers are accessible only in user space and so are visible and relevant only to Tor. We use the UDP transport layer and DTLS to transport TCP packets between the UDP peers. Only part of the TCP packet is transmitted; the details will be discussed in section 3.4, but it serves our purposes now to conceptualize the two nodes as transporting full TCP packets as the UDP datagram’s payload. Upon receiving a UDP datagram, the kernel will remove the UDP header and provide Tor with the enclosed DTLS packet; Tor will decrypt the DTLS payload and present the result (a TCP packet) to its user-level TCP stack. Similarly, when the user-level TCP stack presents a packet for transmission, the node will encrypt it with DTLS and forward the resulting packet to the kernel which then sends it to the

intended destination over UDP. The stack also performs retransmission and acknowledgement of TCP data that are integral to TCP's reliability; these are protected with DTLS and forwarded over UDP in the same manner.

A user-level TCP stack provides an implementation of the suite of socket function calls, such as *socket()*, *send()*, and *recv()*. These reimplementations exist in harmony with the proper set of operating system commands, allowing both a user-level and kernel-level network layer. Thus, data structures and file descriptors created by calls to the user-level stack are visible and relevant only to the parent process; the operating system manages its sockets separately. The user-level stack responds to socket calls by generating packets internally for dispatching as dictated by TCP.

It may seem cumbersome to include an entire TCP implementation as a core component of Tor. In particular, patching the kernel's implementation of TCP to support our features would take significantly less effort. However, Tor relies on volunteers to route traffic; complicated installation procedures are an immediate roadblock towards the ubiquitous use of Tor. The diverse operating systems Tor aims to support and the diverse skill level of its users prevent its installation from requiring external procedures, or even superuser privileges.

Daytona [11] is a user-level TCP stack that we chose for our purposes. It was created by researchers studying network analysis, and consists of the implementation of Linux's TCP stack and the reimplementations of user-level socket functions. It uses *libpcap* to capture packets straight from the Ethernet device and a raw socket to write generated packets, including headers, onto the network. Daytona was designed to operate over actual networks while still giving user-level access to the network implementation. In particular, it allowed the researchers to tune the implementation while performing intrusive measurements. A caveat—there are licensing issues for Daytona's use in Tor. As a result, the deployment of this transport layer into the real Tor network may use a different user-level TCP stack. Our design uses Daytona as a replaceable component and its selection as a user-level TCP stack was out of availability for our proof-of-concept.

3.4 UTCP: Our Tor-Daytona Interface

Our requirements for a user-level TCP stack are to create properly formatted packets, including TCP retransmissions, and to sort incoming TCP/IP packets into data streams: a black box that converts between streams and packets. For our purpose, all notions of routing, Ethernet devices, and interactions with a live network are unnecessary. To access the receiving and transmitting of packets, we commandeer the *r_x*() (receive) and *t_x*()

(transmit) methods of Daytona to instead interface directly with reading and writing to connections in Tor.

UTCP is an abstraction layer for the Daytona TCP stack used as an interface for the stack by Tor. Each UDP connection between ORs has a UTCP-connection object that maintains information needed by our stack, such as the set of circuits between those peers and the socket that listens for new connections. Each circuit has a UTCP-circuit object for similar purposes, such as the local and remote port numbers that we have assigned for this connection.

As mentioned earlier, only part of the TCP header is transmitted using Tor—we call this header the TORTP header; we do this simply to optimize network traffic. The source and destination addresses and ports are replaced with a numerical identifier that uniquely identifies the circuit for the connection. Since a UDP/IP header is transmitted over the actual network, Tor is capable of performing a connection lookup based on the address of the packet sender. With the appropriate connection, and a circuit identifier, the interface to Daytona is capable of translating the TORTP header into the corresponding TCP header.

When the UTCP interface receives a new packet, it uses local data and the TORTP headers to create the corresponding TCP header. The resulting packet is then injected into the TCP stack. When Daytona's TCP stack emits a new packet, a generic *tx*() method is invoked, passing only the packet and its length. We look up the corresponding UTCP circuit using the addresses and ports of the emitted TCP header, and translate the TCP header to our TORTP header and copy the TCP payload. This prepared TORTP packet is then sent to Tor, along with a reference to the appropriate circuit, and Tor sends the packet to the destination OR over the appropriate DTLS connection.

3.5 Congestion Control

The congestion control properties of the new scheme will inherit directly from those of TCP, since TCP is the protocol being used internally. While it is considered an abuse of TCP's congestion control to open multiple streams between two peers simply to send more data, in this case we are legitimately opening one stream for each circuit carrying independent data. When packets are dropped, causing congestion control to activate, it will only apply to the single stream whose packet was dropped. Congestion control variables are not shared between circuits; we discuss the possibility of using the message-oriented Stream Control Transport Protocol (SCTP), which shares congestion control information, in Section 5.2.

If a packet is dropped between two ORs communicat-

ing with multiple streams of varying bandwidth, then the drop will be randomly distributed over all circuits with a probability proportional to their volume of traffic over the link. High-bandwidth streams will see their packets dropped more often and so will back off appropriately. Multiple streams will back off in turn until the congestion is resolved. Streams such as ssh connections that send data interactively will always be allowed to have at least one packet in flight regardless of the congestion on other circuits.

Another interesting benefit of this design is that it gives Tor direct access to TCP parameters at runtime. The lack of sophistication in Tor's own congestion control mechanism is partially attributable to the lack of direct access to networking parameters at the kernel level. With the TCP stack in user space Tor's congestion control can be further tuned and optimized. In particular, end-to-end congestion control could be gained by extending our work to have each node propagate its TCP rate backwards along the circuit: each node's rate will be the minimum of TCP's desired rate and the value reported by the subsequent node. This will address congestion imbalance issues where high-bandwidth connections send traffic faster than it can be dispatched at the next node, resulting in data being buffered upon arrival. When TCP rates are propagated backwards, then the bandwidth between two ORs will be prioritized for data whose next hop has the ability to immediately send the data. Currently there is no consideration for available bandwidth further along the circuit when selecting data to send.

4 Experimental Results

In this section we perform experiments to compare the existing Tor transport layer with an implementation of our proposed TCP-over-DTLS transport. We begin by timing the new sections of code to ensure that we have not significantly increased the computational latency. Then we perform experiments on a local Tor network of routers, determining that our transport has indeed addressed the cross-circuit interference issues previously discussed.

4.1 Timing Analysis

Our UDP implementation expands the datapath of Tor by adding new methods for managing user-level TCP streams and UDP connections. We profile our modified Tor and perform static timing analysis to ensure that our new methods do not degrade the datapath unnecessarily. Experiment 2 was performed to profile our new version of Tor.

The eightieth percentile of measurements for Experiment 2 are given in Table 3. Our results indicate that no

Experiment 2 Timing analysis of our modified TCP-over-DTLS datapath.

- 1: TCP-over-DTLS Tor was modified to time the duration of the aspects of the datapath:
 - injection of a new packet (DTLS decryption, preprocessing, injecting into TCP stack, possibly sending an acknowledgment),
 - emission of a new packet (header translation, DTLS encryption, sending packet),
 - the TCP timer function (increments counters and checks for work such as retransmissions and sending delayed acknowledgements), and
 - the entire datapath from reading a packet on a UDP socket, demultiplexing the result, injecting the packet, reading the stream, processing the cell, writing the result, and transmitting the generated packet.
 - 2: The local Tor network was configured to use 50 ms of latency between connections.
 - 3: A client connected through Tor to request a data stream.
 - 4: Data travelled through the network for several minutes.
-

new datapath component results in a significant source of computational latency.

We have increased the datapath latency to an expected value of 250 microseconds per OR, or 1.5 milliseconds for a round trip along a circuit of length three. This is still an order of magnitude briefer than the round-trip times between ORs on a circuit (assuming geopolitically diverse circuit selection). Assuming each packet is 512 bytes (the size of a cell—a conservative estimate as our experiments have packets that carry full dataframes), we have an upper bound on throughput of 4000 cells per second or 2 MB/s. While this is a reasonable speed that will likely not form a bottleneck, Tor ORs that are willing to devote more than 2 MB/s of bandwidth may require better hardware than the Thinkpad R60 used in our experiments.

4.2 Basic Throughput

We perform Experiment 3 to compare the basic throughput and latency of our modification to Tor, the results of which are shown in Table 4. We can see that the UDP version of Tor has noticeably lower throughput. Originally it was much lower, and increasing the throughput up to this value took TCP tuning and debugging the user-level TCP stack. In particular, errors were uncovered in Daytona's congestion control implementation, and it is

Datapath Component	Duration
Injecting Packet	100 microseconds
Transmitting Packet	100 microseconds
TCP Timer	85 microseconds
Datapath	250 microseconds

Table 3: Time durations for new datapath components. The results provided are the 80th percentile measurement.

Configuration	Network Throughput	Circuit Delay	Base Delay
TCP Tor	176 ± 24.9 KB/s	1026 ± 418 ms	281 ± 12 ms
TCP-over-DTLS Tor	111 ± 10.4 KB/s	273 ± 31 ms	260 ± 1 ms

Table 4: Throughput and delay for different reordering configurations. The configuration column shows which row correspond to which version of Tor we used for our ORs in the experiment. Network throughput is the average data transfer rate we achieved in our experiment. Circuit delay is the latency of the circuit while the large bulk data transfer was occurring, whereas the base delay is the latency of the circuit taken in the absence of any other traffic.

suspected that more bugs remain to account for this disparity. While there may be slight degradation in performance when executing TCP operations in user space instead of kernel space, both implementations of TCP are based on the same Linux TCP implementation operating over in the same network conditions, so we would expect comparable throughputs as a result. With more effort to resolve outstanding bugs, or the integration of a user-level TCP stack better optimized for Tor’s needs, we expect the disparity in throughputs will vanish. We discuss this further in the future work section.

More important is that the circuit delay for a second stream over the same circuit indicates that our UDP version of Tor vastly improves latency in the presence of a high-bandwidth circuit. When one stream triggers the congestion control mechanism, it does not cause the low-bandwidth client to suffer great latency as a consequence. In fact, the latency observed for TCP-over-DTLS is largely attributable to the base latency imposed on connections by our experimental setup. TCP Tor, in contrast, shows a three-and-a-half fold increase in latency when the circuit that it multiplexes with the bulk stream is burdened with traffic.

The disparity in latency for the TCP version means that information is leaked: the link between the last two nodes is witnessing bulk transfer. This can be used as a reconnaissance technique; an entry node, witnessing a bulk transfer from an client and knowing its next hop, can probe potential exit nodes with small data requests to learn congestion information. Tor rotates its circuits every ten minutes. Suppose the entry node notices a bulk transfer when it begins, and probes various ORs to determine the set of possible third ORs. It could further reduce this set by re-probing after nine minutes, after which time

most of the confounding circuits would have rotated to new links.

We conclude that our TCP-over-DTLS, while currently suffering lower throughput, has successfully addressed the latency introduced by the improper use of the congestion control mechanism. We expect that once perfected, the user-level TCP stack will have nearly the same throughput as the equivalent TCP implementation in the kernel. The response latency for circuits in our improved Tor is nearly independent of throughput on existing Tor circuits travelling over the same connections; this improves Tor’s usability and decreases the ability for one circuit to leak information about another circuit using the same connection through interference.

4.3 Multiplexed Circuits with Packet Dropping

Packet dropping occurs when a packet is lost while being routed through the Internet. Packet dropping, along with packet reordering, are consequences of the implementation of packet switching networks and are the prime reason for the invention of the TCP protocol. In this section, we perform an experiment to contrast the effect of packet dropping on the original version of Tor and our improved version.

We reperformed Experiment 1—using our TCP-over-DTLS implementation of Tor instead of the standard implementation—to investigate the effect of packet dropping. The results are presented in Tables 5 and 6. We reproduce our results from Tables 1 and 2 to contrast the old (TCP) and new (TCP-over-DTLS) transports.

We find that throughput is much superior for the TCP-over-DTLS version of Tor. This is likely because the

Version	Configuration	Network Throughput (KB/s)	Circuit Throughput (KB/s)	Throughput Degradation	Effective Drop Rate
TCP-over-DTLS	No dropping	284 ± 35	47.3 ± 5.8	0 %	0 %
	0.1 % (remain.)	261 ± 42	43.5 ± 7.0	8 %	0.08 %
	0.1 % (shared)	270 ± 34	45.2 ± 5.6	4 %	0.03 %
TCP	No dropping	221 ± 6.6	36.9 ± 1.1	0 %	0 %
	0.1 % (remain.)	208 ± 14	34.7 ± 2.3	6 %	0.08 %
	0.1 % (shared)	184 ± 17	30.8 ± 2.8	17 %	0.03 %

Table 5: Throughput for different dropping configurations.

Version	Configuration	Average Latency	Latency Degradation	Effective Drop Rate
TCP-over-DTLS	No dropping	428 ± 221 ms	0 %	0 %
	0.1 % (remaining)	510 ± 377 ms	20 %	0.08 %
	0.1 % (shared)	461 ± 356 ms	7 %	0.03 %
TCP	No dropping	933 ± 260 ms	0 %	0 %
	0.1 % (remaining)	983 ± 666 ms	5.4 %	0.08 %
	0.1 % (shared)	1053 ± 409 ms	12.9 %	0.03 %

Table 6: Latency for different dropping configurations.

TCP congestion control mechanism has less impact on throttling when each TCP stream is separated. One stream may back off, but the others will continue sending, which results in a greater throughput over the bottleneck connection. This is reasonable behaviour since TCP was designed for separate streams to function over the same route. If congestion is a serious problem then multiple streams will be forced to back off and find the appropriate congestion window. Importantly, the streams that send a small amount of data are much less likely to need to back off, so their small traffic will not have to compete unfairly for room inside a small congestion window intended to throttle a noisy connection. The benefits of this are clearly visible in the latency as well: cells can travel through the network considerably faster in the TCP-over-DTLS version of Tor. Despite the large confidence intervals for latency mentioned earlier, we see now that TCP-over-DTLS consistently has significantly lower latency than the original TCP Tor.

The TCP-over-DTLS version has its observed throughput and latency affected proportionally to packet drop rate. It did not matter if the drop was happening on the shared link or the remaining link, since the shared link is not a single TCP connection that multiplexes all traffic. Missing cells for different circuits no longer cause unnecessary waiting, and so the only effect on latency and throughput is the effect of actually dropping cells along circuits.

5 Alternative Approaches

There are other means to improve Tor’s observed latency than the one presented in this paper. For comparison, in this section we outline two significant ones: UDP-OR, and SCTP-over-DTLS.

5.1 UDP-OR

Another similar transport mechanism for Tor has been proposed by Viecco [15] that encapsulates TCP packets from the OP and sends them over UDP until they reach the exit node. Reliability guarantees and congestion control are handled by the TCP stacks on the client and the exit nodes, and the middle nodes only forward traffic. A key design difference between UDP-OR and our proposal is that ours intended on providing backwards compatibility with the existing Tor network while Viecco’s proposal requires a synchronized update of the Tor software for all users. This update may be cumbersome given that Tor has thousands of routers and an unknown number of clients estimated in the hundreds of thousands.

This strategy proposes benefits in computational complexity and network behaviour. Computationally, the middle nodes must no longer perform unnecessary operations: packet injection, stream read, stream write, packet generation, and packet emission. It also removes the responsibility of the middle node to handle retrans-

Experiment 3 Basic throughput and delay for TCP and TCP-over-DTLS versions of Tor.

- 1: To compare TCP and TCP-over-DTLS we run the experiment twice: one where all ORs use the original TCP version of time, and one where they all use our modified TCP-over-DTLS version of Tor.
 - 2: A local Tor network running six routers on a local host was configured to have a latency of 50 milliseconds.
 - 3: Two OPs are configured to connect to our local Tor network. They use distinct circuits, but each OR along both circuit is the same. The latency-OP will be used to measure the circuit's latency by sending periodic timestamp probes over Tor to a timing server. The throughput-OP will be used to measure the circuit's throughput by requesting a large bulk transfer and recording the rate at which it arrives.
 - 4: We start the latency-OP's timestamp probes and measure the latency of the circuit. Since we have not begun the throughput-OP, we record the time as the base latency of the circuit.
 - 5: We begin the throughput-OP's bulk transfer and measure throughput of the circuit. We continue to measure latency using the latency-OP in the presence of other traffic. The latency results that are collected are recorded separately from those of step 4.
 - 6: Data was collected for over a minute, and each configuration was run a half dozen times to obtain confidence intervals.
-

missions, which means a reduction in its memory requirements. The initial endpoint of communication will be responsible for retransmitting the message if necessary. We have shown that computational latency is insignificant in Tor, so this is simply an incidental benefit.

The tangible benefit of UDP-OR is to improve the network by allowing the ORs to function more exactly like routers. When cells arrive out of order at the middle node, they will be forwarded regardless, instead of waiting in input buffers until the missing cell arrives. Moreover, by having the sender's TCP stack view both hops as a single network, we alleviate problems introduced by disparity in network performance. Currently, congestion control mechanisms are applied along each hop, meaning that an OR in the middle of two connections with different performance metrics will need to buffer data to send over the slower connection. Tor provides its own congestion control mechanism, but it does not have the sophistication of TCP's congestion control.

We require experimentation to determine if this proposal is actually beneficial. While it is clear that memory requirements for middle nodes are reduced [15], the endpoints will see increased delay for acknowledge-

ments. We expect an equilibrium for total system memory requirements since data will be buffered for a longer time. Worse, the approach shifts memory requirements from being evenly distributed to occurring only on exit nodes—and these nodes are already burdened with extra responsibilities. Since a significant fraction of Tor nodes volunteer only to forward traffic, it is reasonable to use their memory to ease the burden of exit nodes.

Circuits with long delays will also suffer reduced throughput, and so using congestion control on as short a path as possible will optimize performance. If a packet is dropped along the circuit, the endpoint must now generate the retransmission message, possibly duplicating previous routing efforts and wasting valuable volunteered bandwidth. It may be more efficient to have nodes along a circuit return their CWND for the next hop, and have each node use the minimum of their CWND and the next hop's CWND. Each node then optimizes their sending while throttling their receiving.

5.1.1 Low-cost Privacy Attack

UDP-OR may introduce an attack that permits a hostile entry node to determine the final node in a circuit. Previously each OR could only compute TCP metrics for ORs with whom they were directly communicating. Viecco's system would have the sender's TCP stack communicate indirectly with an anonymous OR. Connection attributes, such as congestion and delay, are now known for the longer connection between the first and last nodes in a circuit. The first node can determine the RTT for traffic to the final node. It can also reliably compute the RTT for its connection to the middle node. The difference in latency reflects the RTT between the second node and the anonymous final node. An adversary can use a simple technique to estimate the RTT between the second node and every other UDP Tor node in the network [3], possibly allowing them to eliminate many ORs from the final node's anonymity set. If it can reduce the set of possible final hops, other reconnaissance techniques can be applied, such as selectively flooding each OR outside of Tor and attempting to observe an increased latency inside Tor [10]. Other TCP metrics may be amalgamated to further aid this attack: congestion window, slow-start threshold, occurrence of congestion over time, standard deviation in round-trip times, etc. The feasibility of this attack should be examined before allowing nodes who do not already know each other's identities to share a TCP conversation.

5.2 Stream Control Transmission Protocol

The Stream Control Transmission Protocol (SCTP) [13] is a message-based transport protocol. It provides sim-

ilar features to TCP: connection-oriented reliable delivery with congestion control. However, it adds the ability to automatically delimit messages instead of requiring the receiving application to manage its own delimiters. The interface is based on sending and receiving messages rather than bytes, which is appropriate for Tor's cell-based transport.

More importantly, SCTP also adds a feature well-suited to our purposes—multiple streams can be transported over the same connection. SCTP allows multiple independent ordered streams to be sent over the same socket; we can use this feature to assign each circuit a different stream. Cells from each circuit will arrive in the order they were sent, but the order cells arrive across all circuits may vary from their dispatch order. This is exactly the behaviour we want for cells from different circuits being sent between the same pair of ORs.

While SCTP is not as widely deployed as TCP, the concept of using a user-level SCTP implementation [5] inside Tor remains feasible. This suggests a SCTP-over-DTLS transport similar in design to our TCP-over-DTLS design. This means that the extra benefits of TCP-over-DTLS will also extend to SCTP-over-DTLS: backwards compatibility with the existing Tor network, a constant number of kernel-level sockets required, and a secured transport header.

What is most interesting about the potential of SCTP-over-DTLS is SCTP's congestion control mechanism. Instead of each TCP stream storing its own congestion control metrics, SCTP will share metrics and computations across all streams. An important question in the development of such a scheme is whether SCTP will act fairly towards streams that send little data when other streams invoke congestion control, and whether the sharing of congestion control metrics results in a privacy-degrading attack by leaking information.

6 Future Work

6.1 Live Experiments

The most pressing future work is to perform these experiments on live networks of geographically distributed machines running TCP-over-DTLS Tor, using computers from the PlanetLab network, or indeed on the live Tor network. Once running, we could measure latency and throughput as we have already in our experiments, comparing against results for regular Tor. Moreover, we can also compare other approaches, such as SCTP-over-DTLS and UDP-OR, using the same experiments. Note that UDP-OR could of course not be tested on the live Tor network, but it could be in a PlanetLab setup. A key metric will be the distribution of throughput and latency for high- and low-volume circuits before and after our im-

provements, and an analysis of the cause of the change. Additionally, once most ORs use UDP, we can determine if the reduced demand on open sockets solves the problem of socket proliferation on some operating systems.

6.2 TCP Stack Memory Management

Tor requires thousands of sockets to buffer fixed-size cells of data, but data is only buffered when it arrives out-of-order or has not been acknowledged. We envision dynamic memory management such as a shared cell pool to handle memory in Tor. Instead of repeatedly copying data cells from various buffers, each cell that enters Tor can be given a unique block of memory from the cell pool until it is no longer needed. A state indicates where this cell currently exists: input TCP buffer, input Tor buffer, in processing, output Tor buffer, output TCP buffer. This ensures that buffers are not allocated to store empty data, which reduces the overall memory requirements. Each cell also keeps track of its socket number, and its position in the linked list of cells for that socket. While each socket must still manage data such as its state and metrics for congestion control, this is insignificant as compared to the current memory requirements. This permits an arbitrary number of sockets, for all operating systems, and helps Tor's scalability if the number of ORs increases by orders of magnitude.

This approach results in the memory requirements of Tor being a function of the number of cells it must manage at any time, independent of the number of open sockets. Since the memory requirements are inextricably tied to the throughput Tor offers, the user can parameterize memory requirements in Tor's configuration just as they parameterize throughput. A client willing to denote more throughput than its associated memory requirements will have its contribution throttled as a result. If network conditions result in a surge of memory required for Tor, then it can simply stop reading from the UDP multiplexing socket. The TCP stacks that sent this unread data will assume there exists network congestion and consequently throttle their sending—precisely the behaviour we want—while minimizing leaked information about the size of our cell pool.

7 Summary

Anonymous web browsing is an important step in the development of the Internet, particularly as it grows ever more inextricable from daily life. Tor is a privacy-enhancing technology that provides Internet anonymity using volunteers to relay traffic, and uses multiple relays in series to ensure that no entity (other than the client) in the system is aware of both the source and destination of messages.

Relaying messages increases latency since traffic must travel a longer distance before it is delivered. However, the observed latency of Tor is much larger than just this effect would suggest. To improve the usability of Tor, we examined where this latency occurs, and found that it happens when data sat idly in buffers due to congestion control. Since multiple Tor circuits are multiplexed over a single TCP connection between routers, we observed cross-circuit interference due to the nature of TCP's in-order, reliable delivery and its congestion control mechanisms.

Our solution was the design and implementation of a TCP-over-DTLS transport between ORs. Each circuit was given a unique TCP connection, but the TCP packets themselves were sent over the DTLS protocol, which provides confidentiality and security to the TCP header. The TCP implementation is provided in user space, where it acts as a black box that translates between data streams and TCP/IP packets. We performed experiments on our implemented version using a local experimentation network and showed that we were successful in removing the observed cross-circuit interference and decreasing the observed latency.

Acknowledgements

We would like to thank Steven Bellovin, Vern Paxson, Urs Hengartner, S. Keshav, and the anonymous reviewers for their helpful comments on improving this paper. We also gratefully acknowledge the financial support of The Tor Project, MITACS, and NSERC.

References

- [1] Tim Dierks and Eric Rescorla. RFC 5246—The Transport Layer Security (TLS) Protocol Version 1.2. <http://www.ietf.org/rfc/rfc5246.txt>, August 2008.
- [2] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The Second-Generation Onion Router. *Proceedings of the 13th USENIX Security Symposium*, 2004.
- [3] Krishna P. Gummadi, Stefan Saroiu, and Steven D. Gribble. King: Estimating Latency between Arbitrary Internet End Hosts. *ACM SIGCOMM Computer Communication Review*, 2002.
- [4] Information Sciences Institute. RFC 793—Transmission Control Protocol. <http://www.ietf.org/rfc/rfc793.txt>, September 1981.
- [5] Andreas Jungmaier, Herbert Hölzlwimmer, Michael Tüxen, and Thomas Dreibholz. The SCTP library (sctplib). <http://www.sctp.de/sctp-download.html>, 2007. Accessed February 2009.
- [6] Stephen Kent and Randall Atkinson. RFC 2401—Security Architecture for the Internet Protocol. <http://www.ietf.org/rfc/rfc2401.txt>, November 1998.
- [7] Marcus Leech et al. RFC 1928—SOCKS Protocol Version 5. <http://www.ietf.org/rfc/rfc1928.txt>, March 1996.
- [8] Damon McCoy, Kevin Bauer, Dirk Grunwald, Parisa Tabriz, and Douglas Sicker. Shining Light in Dark Places: A Study of Anonymous Network Usage. University of Colorado Technical Report CU-CS-1032-07, August 2007.
- [9] Nagendra Modadugu and Eric Rescorla. The Design and Implementation of Datagram TLS. *Network and Distributed System Security Symposium*, 2004.
- [10] Steven J. Murdoch and George Danezis. Low-Cost Traffic Analysis of Tor. In *IEEE Symposium on Security and Privacy*, pages 183–195, 2005.
- [11] Prashant Pradhan, Srikanth Kandula, Wen Xu, Anees Shaikh, and Erich Nahum. Daytona: A User-Level TCP Stack. <http://nms.lcs.mit.edu/~kandula/data/daytona.pdf>, 2002.
- [12] Joel Reardon. Improving Tor using a TCP-over-DTLS Tunnel. Master's thesis, University of Waterloo, Waterloo, ON, September 2008.
- [13] Randall Stewart, Qiaobing Xie, Ken Morneault, Chip Sharp, Hanns Juergen Schwarzbauer, Tom Taylor, Ian Rytina, Malleswar Kalla, Lixia Zhang, and Vern Paxson. RFC 2960—Stream Control Transmission Protocol. <http://www.ietf.org/rfc/rfc2960.txt>, October 2000.
- [14] TorStatus. Tor Network Status. <http://torstatus.kgprog.com/>. Accessed February 2009.
- [15] Camilo Viecco. UDP-OR: A Fair Onion Transport Design. <http://www.petsymposium.org/2008/hotpets/udp-tor.pdf>, 2008.

Locating Prefix Hijackers using LOCK

Tongqing Qiu
Georgia Tech
tongqqiu@cc.gatech.edu

Lusheng Ji
AT&T Labs – Research
lji@research.att.com

Dan Pei
AT&T Labs – Research
peidan@research.att.com

Jia Wang
AT&T Labs – Research
jiawang@research.att.com

Jun (Jim) Xu
Georgia Tech
jx@cc.gatech.edu

Hitesh Ballani
Cornell University
hitesh@cs.cornell.edu

Abstract

Prefix hijacking is one of the top known threats on today's Internet. A number of measurement based solutions have been proposed to detect prefix hijacking events. In this paper we take these solutions one step further by addressing the problem of locating the attacker in each of the detected hijacking event. Being able to locate the attacker is critical for conducting necessary mitigation mechanisms at the earliest possible time to limit the impact of the attack, successfully stopping the attack and restoring the service.

We propose a robust scheme named LOCK, for Locating the prefix hijacker ASes based on distributed Internet measurements. LOCK locates each attacker AS by actively monitoring paths (either in the control-plane or in the data-plane) to the victim prefix from a small number of carefully selected monitors distributed on the Internet. Moreover, LOCK is robust against various countermeasures that the hijackers may employ. This is achieved by taking advantage of two observations: that the hijacker cannot manipulate AS path before the path reaches the hijacker, and that the paths to victim prefix “converge” around the hijacker AS. We have deployed LOCK on a number of PlanetLab nodes and conducted several large scale measurements and experiments to evaluate the performance. Our results show that LOCK is able to pinpoint the prefix hijacker AS with an accuracy up to 94.3%.

1 Introduction

The Internet consists of tens of thousands of Autonomous Systems (ASes), each of which is an independently administrated domain. Inter-AS routing information is maintained and exchanged by the Border Gateway Protocol (BGP). The lack of adequate authentication schemes in BGP leaves an opportunity for misbehaving routers to advertise and spread fabricated AS paths for

targeted prefixes. Originating such a false AS path announcement is referred to as “prefix hijacking”. Once a BGP router accepts such a false route and replaces a legitimate route with it, the traffic destined for the target prefix can be redirected as the hijacker wishes. The victim prefix network of a successful hijacking will experience performance degradation, service outage, and security breach. The incident of the prefix of YouTube being hijacked by an AS in Pakistan for more than 2 hours [1] is just a recent and better known reminder of the possibility of real prefix hijacking attacks.

Recently proposed solutions for combating prefix hijacking either monitor the state of Internet and detect ongoing hijacking events [12, 21, 22, 26, 34, 45], or attempt to restore service for victim prefix networks [42]. Both approaches are compatible with existing routing infrastructures and generally considered more deployable than another family of proposals (e.g., [4, 8, 11, 13, 18, 19, 27, 32, 35–37, 44]) which aim at prefix hijacking prevention, because the latter usually require changes to current routing infrastructures (e.g., router software, network operations), and some also require public key infrastructures.

However, the aforementioned detection and service restoration solutions only solve parts of the problem and a critical step is still missing towards a complete and automated detection-recovery system. That is how to locate the hijackers. More importantly, the location of hijackers is one of the key information that enables existing mitigation methods against prefix hijacking (e.g., [42]). One may consider this step trivial. Indeed in current practice this step is actually accomplished by human interactions and manual inspections of router logs. However, we would argue that the success of the current practice is due to the fact that discovered attacks so far are still primitive. Many of them are simply not attacks but rather the results of router mis-configurations. As we will elaborate, locating sophisticated hijackers is far from a trivial problem and the current practice will have great difficulties in locating them.

In this paper, we present a scheme called LOCK to LOCate prefix hijackKers. It is a light-weight and incrementally deployable scheme for locating hijacker ASes. The main idea behind LOCK are based the following two observations: that the hijacker cannot manipulate AS path before the path reaches the hijacker, and that the paths to victim prefix “converge” around the hijacker AS. Our contributions are four-fold. First, to the best of our knowledge, it is the first work studying the attacker locating problem for prefix hijacking, even when countermeasures are engaged by the hijacker. Second, our locating scheme can use either data-plane or control-plane information, making the deployment more flexible in practice. Third, we propose an algorithm for selecting locations where data-plane or control-plane data are collected such that the hijackers can be more efficiently located. Finally, we have deployed LOCK on a number of PlanetLab nodes and conducted several large scale measurements and experiments to evaluate the performance of LOCK against three groups of hijacking scenarios: synthetic attacks simulated using real path and topology information collected on the Internet, reconstructed previously known attacks, and controlled attack experiments conducted on the Internet. We show that the proposed approach can effectively locate the attacker AS with up to 94.3% accuracy.

The rest of the paper is organized as follows. Section 2 provides background information on prefix hijacking. Section 3 provides an overview of the framework of our LOCK scheme. Then we describe detailed monitoring and locating methodologies in Section 4 and Section 5 respectively. Section 6 evaluates the performance of the LOCK scheme. Section 7 briefly surveys related works before Section 8 concludes the paper.

2 Background

As mentioned before, IP prefix hijacking occurs when a mis-configured or malicious BGP router either originates or announces an AS path for an IP prefix not owned by the router’s AS. In these BGP updates the misbehaving router’s AS appears very attractive as a next hop for forwarding traffic towards that IP prefix. ASes that receive such ill-formed BGP updates may accept and further propagate the false route. As a result the route entry for the IP prefix in these ASes may be *polluted* and traffic from certain part of the Internet destined for the victim prefix is redirected to the attacker AS.

Such weakness of the inter-domain routing infrastructure has great danger of being exploited for malicious purposes. For instance the aforementioned misbehaving AS can either drop all traffic addressed to the victim prefix that it receives and effectively perform a denial-of-service attack against the prefix owner, or redirect traf-

fic to an incorrect destination and use this for a phishing attack [28]. It can also use this technique to spread spams [33].

We refer to this kind of route manipulation as *IP prefix hijack attacks* and the party conducting the attack *hijacker* or *attacker*. Correspondingly the misbehaving router’s AS becomes the *hijacker AS*, and the part of the Internet whose traffic towards the victim prefix is redirected to the hijacker AS is *hijacked*. So do we call the data forwarding paths that are now altered to go through the hijacker AS *hijacked*. We also refer to the victim prefix as the *target prefix*.

Following the convention in [45], we classify prefix hijacks into the following three categories:

- **Blackholing:** the attacker simply drops the hijacked packets.
- **Imposture:** the attacker responds to senders of the hijacked traffic, mimicking the true destination’s (the target prefix’s) behavior.
- **Interception:** the attacker forwards the hijacked traffic to the target prefix after eavesdropping/recording the information in the packets.

While the conventional view of the damage of prefix hijacking has been focused on blackholing, the other two types of hijacking are equally important, if not more damaging [6]. In addition, the characteristics of different hijack types are different, which often affect how different types of attacks are detected. In this paper, we use the term *hijack* to refer to all three kinds of prefix hijack attacks unless otherwise specified.

There have been a number of approaches proposed for detecting prefix hijacks. They utilize either information in BGP updates collected from control plane [21, 22, 26, 34], or end-to-end probing information collected from data plane [43, 45], or both [6, 12, 36]. We will not get into the details of most of these approaches here because LOCK is a *hijacker-locating scheme*, not a hijack detection scheme. The difference between these two will be explained later in section 3.1. To locate a hijacker, the LOCK scheme only needs to know whether a given prefix is hijacked. Therefore LOCK can be used together with any detection method to further locate the hijacker AS. Moreover, LOCK can locate the hijacker using either data-plane or control-plane information.

3 Framework

In this section, we present an overview of key ideas of the hijacker locating algorithm in LOCK. Similar to detecting prefix hijacking, locating hijacker AS can be done in either control-plane or data-plane. Either way, the goal

is, to use the AS path information to the hijacked prefix observed at multiple and diverse vantage points (or monitors) to figure out who the hijacker is. In control-plane approach, the AS path information is obtained from BGP routing tables or update messages. In data-plane approach, the AS path is obtained via AS-level traceroute (mapping the IP addresses in traceroute to AS numbers).

Both methods have pros and cons. Realtime data-plane information from multiple diverse vantage points is easier to be obtained than realtime BGP information (e.g. the BGP updates from [3] are typically delayed for a few hours). On the other hand, it is relatively easier for the attacker to manipulate the data-plane AS path to countermeasure the locating algorithm than the control-plane AS path. LOCK can use either data-plane or control-plane AS paths to locate the hijackers.

3.1 Challenges

Currently, the most commonly used hijacker-locating approach (called *simple locating approach*) is to look at the origin ASes of the target prefix. For example, Figure 1 (a) shows the control-plane AS path information to target prefix p at vantage points $M1$, $M2$, and $M3$, respectively, before hijacker H launches the hijack. All three vantage points observe the origin AS is T . In Figure 1 (b), Hijacker AS H announces a path H to target prefix p , which ASes A , B , $M1$, and $M2$ accept since the paths via H are better than their previous ones via CDT . In this case, the simple locating approach can easily identify the newly-appearing origin AS H as the hijacker.

However, this simple locating approach can fail even without any countermeasures by the hijackers. For example, in Figure 1(c), hijacker H pretends there is a link between H and the target AS T , and announces an AS path HT , again accepted by $A, B, M1$, and $M2$. The simple locating approach does not work here since the origin AS in all the AS paths are still T .

One might try to look beyond just origin AS and check other ASes in the path, but the hijacker AS might counter this such that the hijacker AS might not even appear in any of the AS paths. For example, in Figure 1(d) H simply announces an AS path T without prepending its own AS number H ¹.

Above challenges in control-plane locating approaches also exist in data-plane approaches. Almost all data-plane path probing mechanisms are derived from the well known *traceroute* program. In *traceroute*, triggering messages with different initial TTL values are sent towards the same destination. As these messages are forwarded along the path to this destination, as soon as a message's TTL reduces to zero after reaching a router, the router needs to send back an ICMP Timeout message to notify the probing source. If the triggering messages

go through the hijacker, this happens when the triggering messages' initial TTL values are greater than the hop distance from the probing source to the hijacker, the hijacker can do many things to interfere the path probing as a countermeasure to the locating algorithm.

In Figure 2(a), the hijacker AS's border router responds to traceroute honestly in blackholing (in which for example the border router responds with a *no route* ICMP message with its own IP address) and imposture (in which for example a router in H responds "destination reached" message with its own IP address). In either case, the router address belongs to H and maps to AS H , and the simple locating approach can identify H as the newly appearing origin AS hence the hijacker AS.

However, in the interception attack shown in Figure 2(b), the hijacker further propagates the traceroute probe packets to the target via $XYZT$, thus the origin AS is still H . Hence the simple locating approach fails in this case.

Furthermore, the hijacker can use various countermeasures. For instance, the hijacker may simply drop the triggering messages without replying to interrupt *traceroute* probing from proceeding further. Or it may send back ICMP Timeout messages with arbitrary source IP addresses to trick the probing source into thinking routers of those addresses are en route to the destination. The hijacker may even respond with ICMP Timeout messages before the triggering messages' TTL values reach zero. In Figure 2 (c), hijacker H manipulates the traceroute response such that after the IP-to-AS mapping, the AS path appears to $M1$ to be $ACDT$, and appears to $M2$ to be BDT , neither of which contains hijacker AS H in it, making the hijacker locating difficult. We refer to above manipulation of traceroute response as *countermeasure* for data-plane locating approach, and call such hijackers *countermeasure-capable* or *malicious*.

In summary, sophisticated hijackers that are capable of engaging countermeasures can inject false path information into measurements collected in both control plane and data plane, easily evading simple hijacker-locating mechanisms. We therefore design a more effective algorithm for locating these hijackers in the next section.

3.2 Locating Hijackers

The basic idea of LOCK is based on two key observations, which apply to both data-plane and control-plane approaches, different types of hijacks (blackholing, imposture, and interception), and with or without countermeasures by the attackers.

The first observation is that *the hijacker cannot manipulate the portion of the AS path from a polluted vantage point to the upstream (i.e., closer to the vantage point) neighbor AS of the hijacker AS*. For example, in

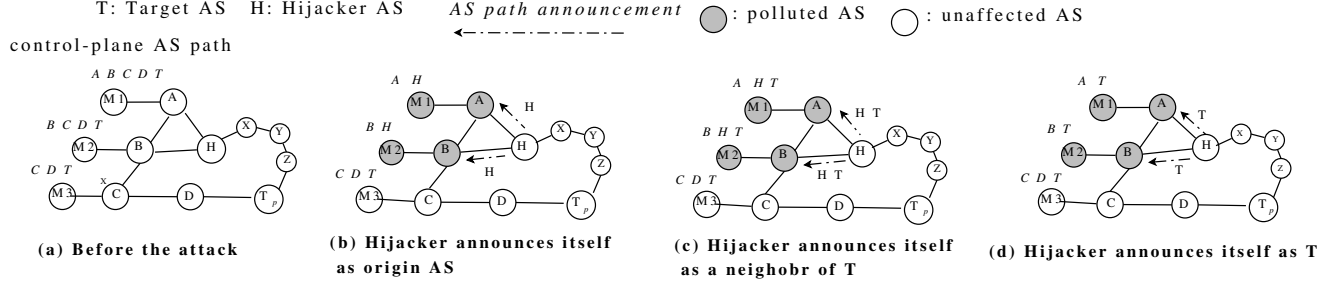


Figure 1: Control plane examples

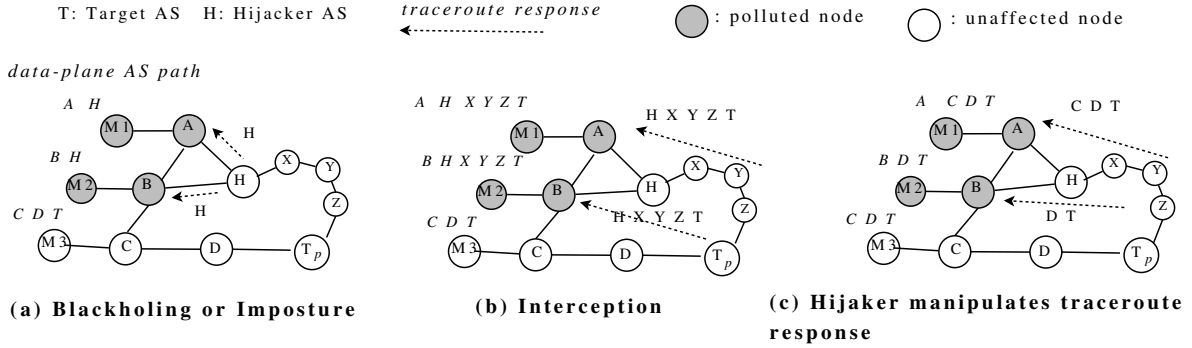


Figure 2: Data plane examples

Figures 1(c) and (d) and Figures 2 (b) and (c), for the polluted vantage points $M1$ and $M2$, the upstream ASes for hijacker AS H are A and B , and the portion of AS path $M1A$ and $M2B$ are trustworthy. This is easy to understand since the routers from the vantage points to hijacker upstream ASes are all well-behaving ones thus conform to BGP protocol in control-plane and ICMP protocol used in traceroute in data-plane.

The second observation is that *the trustworthy portion of polluted AS paths from multiple vantage points to a hijacked victim prefix “converge” “around” the hijacker AS*. This is also intuitive since, if the set of monitors are topologically diverse enough, the trustworthy portion of AS paths from all the polluted monitors to the target prefix must include the upstream AS neighbors of the hijacker AS (e.g. in Figure 1(d), and Figure 2 (c)) thus converge “around” the hijacker AS, or directly converge at hijacker AS (e.g. in Figure 1(b) and (c) and Figure 2(a) and (b)).

Since we do not know beforehand the hijack scenarios and whether there is any countermeasure, we focus on identifying these upstream neighbors of the hijacker AS, and then intuitively hijacker should be within the intersection of the 1-hop neighbor sets of the hijacker’s neighbors. And chances are that the size of the intersection set is very small if the monitors have diversified locations. The neighbor sets of a given AS can be obtained from

a daily snapshot of the state-of-arts AS level topology repository such as [16].

For example, in both Figures 1 and 2, ideally suppose we know that ASes A, B (which are on the polluted paths from vantage points $M1$ and $M2$, respectively) are the upstream neighbors of the hijacker. We can then infer that the hijacker AS should be within the intersection of $neighborset(A) = \{M1, B, H\}$ and $neighborset(B) = \{M2, A, H\}$, which is H . Of course in reality LOCK does not know beforehand which ASes are the upstream neighbors of the hijackers, thus each AS in a polluted path can potentially be such a neighbor of the hijacker AS. And hence the hijacker could be a neighbor of any of these nodes. We therefore put all the neighbors of *each* AS on a polluted path together with the path nodes themselves to form a *neighborhood set* of the polluted path. The hijacker should be included in this neighborhood set.

For reasons that we will explain in the Section 5, instead of using the neighborhood set of an arbitrary path, LOCK conservatively starts from the union of all the neighborhood sets of all polluted paths, \mathcal{H} . Then given that all polluted paths go through a neighbor AS of the hijacker, an AS which appears in more neighborhood sets is more likely to be the hijacker. We thus “rank” the ASes within \mathcal{H} based on how many neighborhood sets an AS is in to narrow down to the handful of top ranked ASes.

Also when there are multiple convergence points, the earliest convergence point is more likely to be the hijacker than the later ones. More detailed ranking algorithm will be presented in Section 5.

As shown in this section, LOCK can utilize either control-plane or data-plane information. However, for the ease of presentation and due to space limitation, in the rest of paper we focus on data-plane approach unless otherwise specified.

4 Monitor Selection

LOCK operates in a distributed fashion from a number of monitors on the Internet. Both the number of monitors and locations of these monitors affect the accuracy in locating prefix hijackers. In general, the more monitors used by LOCK, the higher accuracy LOCK can achieve in locating prefix hijackers, and the more measurement overhead are incurred by LOCK. More importantly, the measurement overhead increase linearly as the number of monitors increases, while at the same time the improved accuracy gained by each additional monitor can gradually diminish. Therefore, it is hopeful to achieve very good accuracy with a limited number of carefully selected monitors.

In this section, we present a novel algorithm for selecting a number of monitors from a candidate set. In particular, we model the monitor selection problem as follows. Initially, we have M candidate monitors around the world. For each target prefix, we select a subset m monitors among the M candidates. In order to achieve the highest possible hijacker-locating accuracy with a limited number of monitors, the selection of monitors should be guided by two objectives: (i) maximize the likelihood of observing hijacking events on the target prefix; and (ii) maximize the diversity of paths from monitors to the target prefix so that a hijacking event can be observed from multiple distinct vantage points.

Our monitor selection algorithm consists of three steps:

1. **Clustering:** The M candidate monitors are grouped into m clusters. Monitors in the same cluster have more similar paths to the target prefix than those in different clusters.
2. **Ranking:** Candidate monitors in each cluster are ranked based on probability of their paths to the target prefix being polluted when the prefix is hijacked. The monitors with higher ranks are more likely to observe the hijacking event.
3. **Selecting:** The monitor which ranks the highest in each cluster is chosen to monitor the target prefix.

Thus, a total of m monitors are selected for each target prefix.

4.1 Clustering

For a given target prefix, the candidate monitors are clustered based on similarity of their AS-level paths to the prefix. We measure the *similarity* between a pair of paths as the number of common ASes between these two paths over the length of the shorter path. If there is no common AS, the similarity score is 0. On the other hand, if the two paths are identical or one path is a sub-path of the other, the similarity score is 1. We also define the *similarity* between two clusters of paths as the maximum similarity between any two paths, one from each cluster.

We model the clustering part as a hierarchical clustering problem. Such problems have well-known algorithms, such as [17], that are polynomial-time complex. In this paper, we adopt the following simple clustering algorithm². First, we start from M clusters, with one candidate site in each cluster, and compute similarity score for each pair of clusters. Second, we identify the pair of clusters with the largest similarity score among all pairs of clusters, and merge these two clusters into a single cluster. Third, we recompute the similarity score between this newly-formed cluster with each of the other clusters. We repeat steps two and three until only m clusters remain.

4.2 Ranking

We rank candidate monitors in each cluster based on their likelihood of observing hijacking events on the target prefix t (i.e., the path from monitor to target prefix is polluted by hijacking). For a given candidate site s , whether or not the route from s to t is polluted by hijacker h depends on the original best route (before the hijacking happens) from s to t and the fake route announced by h . This has been demonstrated by previous analysis in [6].

We assume that “prefer customer route” and “valley-free routing” are commonly adopted interdomain routing policies on today’s Internet. We denote the original best route from s to t as a “customer-route”, a “peer-route”, or a “provider-route” if the next-hop AS on the route from s to t is a customer, a peer, or a provider of the AS to which s belongs, respectively. According to the interdomain routing policies, a customer-route would be the most preferable and a provider-route would be the least preferable by each router; similarly, when policy preferences are equal, the route with shorter AS path is more preferable [10]. Therefore, when hijacker h announces a fake path, the monitor whose original best route is provider-route is more likely to be polluted than a original route of peer-route, which in turn is more likely to be polluted

Algorithm 1: Ranking monitors in each cluster

```
1 foreach monitor  $i$  in the cluster
2   if provider-route  $R[i] = 300$ ; /* Assign the
   ranking. The larger the number is, the higher the
   rank is. */
3   elseif peer-route  $R[i] = 200$ ;
4   else  $R[i] = 100$ ;
5    $R[i] += D(i, t)$ ; /* Add the AS-level distance */
```

than a original route of customer-route; when the policy preferences are equal, the monitor whose original best route has a longer AS path to t is more likely to be polluted than the one whose original best route has a shorter AS path (Please refer to Table 1 of [6] for detailed analysis). Our ranking algorithm is shown in Algorithm 1. Note that establishing AS topology itself is a challenging problem. We use most advanced techniques [30] to infer the AS relationship. Admittedly, inferred results could be incomplete. However, the evaluation part will show that the ranking algorithm based on such data can still achieve high location accuracy.

5 Hijacker-Locating Algorithm

LOCK locates hijacker AS based on AS paths from a set of monitors to the victim prefix. The AS path from a monitor to the victim prefix can be either obtained from the control plane (e.g., BGP AS path) or from the data plane (e.g., traceroute path). In the latter case, LOCK will need to pre-process the path and compute the corresponding AS path (described in Section 5.1).

5.1 Pre-Processing

When a prefix is hijacked, a portion of the Internet will experience the hijack. Traffic originated from this portion of the Internet and destined for the hijacked prefix will be altered to go through the hijacker. Monitors deployed in this affected portion of the Internet can observe that their monitor-to-prefix paths being altered. These monitor-to-prefix paths are the foundation of our hijacker-locating algorithm. Only paths changed by the hijack event should be supplied to the hijacker-locating algorithm. Methods such as the one outlined in [45] help separate real hijack induced path changes from changes caused by other non-hijack reasons.

If the monitor-to-prefix path is obtained from the data plane, then LOCK pre-processes the path in the following way. The most common tool for acquiring IP forwarding path in the data plane is the well known *traceroute* program. This program sends out a series of triggering packets with different initial TTL values to trig-

ger the routers en route to the destination to return ICMP Timeout messages as soon as they observe a triggering message's TTL value reaching 0, hence revealing these routers' identities. These *traceroute* results are router-level paths and they need to be converted to AS-level paths. During this conversion, NULL entries in *traceroute* results are simply discarded. This simplification rarely has any effect on the resulted AS path because as *traceroute* proceeds within a particular AS, only if all routers in this AS failed to show up in *traceroute* results our results may be affected, which we have found this to be very rare. These resulting AS paths are known as the "reported paths" by the monitors in the rest of the section.

We use publicly available IP to AS mapping data provided by the iPlane services [15] to convert router IP addresses to their corresponding AS numbers. It is known that accurately mapping IP addresses to AS numbers is difficult due to problems such as Internet Exchange Points (IXPs) and sibling ASes [6, 25]. We argue that the impact of these mapping errors on the results of our hijacker-locating algorithm is minor. Firstly the distribution of the nodes, either routers or ASes, that may cause any mapping error in their corresponding Internet topologies, either router level or AS level, is sparse. If our paths do not contain these problematic nodes, our results are not affected by mapping errors. Secondly, it will become apparent, as more of the details of the hijacker-locating algorithm are described, that our algorithm is rather robust against such mapping errors. As long as these errors do not occur when mapping nodes near the hijacker, they will not affect the result of our algorithm. It is also worthwhile noting that the IP to AS mapping data do not need to be obtained from realtime control plane data. That is, the IP to AS mapping can be pre-computed and stored since it usually does not change over short period of time.

It is also helpful to perform sanity checks on the AS paths before we begin the hijacker-locating algorithm. The hijacker may forge *traceroute* results if a *traceroute* triggering message actually passes through the hijacker. Since the prefix has been hijacked, triggering messages with large enough initial TTL values, at least larger than the hop distance between the probing monitor and the hijacker, will inevitably pass through the hijacker. For a sophisticated hijacker, this is a good opportunity to fabricate responses to these triggering messages to conceal its own identity. As a result, the AS paths mapped from such a fake *traceroute* results may contain erroneous ASes as well. It is easy to see that these "noises" only appear in the later portion of a path because the portion that is before the hijacker cannot be altered by the hijacker, – the ICMP triggering messages do not reach the hijacker. Hence if a node in a path is determined to be a fake node, we really do not need to consider any nodes beyond

that point because this point must be already beyond the hacker's position in the path.

In the pre-processing part, we consider the duplicated appearances of AS nodes. If a node appears more than once in a path, any appearance beyond the first is considered fake. This is because real *traceroute* results should not contain loops.

5.2 Basic Algorithm

We denote the set of monitors that have detected the hijacking and reported their altered monitor-to-prefix paths by \mathcal{M} . For each monitor m_i within \mathcal{M} , there is an AS level monitor-to-prefix path P_i , either computed by pre-processing *traceroute* path or obtained directly from BGP routes. We define the neighborhood set of a specific path P_i , denoted as $\mathcal{N}(P_i)$, as the union of all path nodes and their one-hop neighbors. The target prefix' AS should be removed from all $\mathcal{N}(P_i)$. The reason is simple, – it is not the hijacker AS. Note that LOCK computes the neighborhood set based on AS topology inferred from RouteView [3] before the hijacking is detected, rather than real-time BGP data when the hijacking is ongoing. Though the hijacker can try to pollute the AS topology information before launching real hijacking attack on the victim prefix, the impact of such evasion is minimal on the neighborhood set computation because it is difficult for hijacker to “remove” an observed true link from the AS topology by announcing fake routes.

We are interested in the neighborhood sets of the AS paths instead of just the AS paths themselves because the hijacker may actually not show up in any of the AS paths if it manipulates *traceroute* results. However, even under this condition the ASes which are immediately before the hijacker along the paths are real. Thus, the union of all neighborhood sets of all reported AS paths, $\mathcal{H} = \bigcup_i \mathcal{N}(P_i)$, form our search space for the hijacker. We denote each node in this search space as a_k . The hijacker-locating algorithm is essentially a ranking algorithm which assigns each node in \mathcal{H} a rank based on their suspicious level of being the hijacker.

The LOCK algorithm ranks each AS node $a_k \in \mathcal{H}$ based on two values, *covered count* $\mathcal{C}(a_k)$ and *total distance to monitors* $\mathcal{D}(a_k)$. The *covered count* is simply computed by counting a_k appearing in how many path neighborhood sets. For each neighborhood set $\mathcal{N}(P_i)$ that a_k is a member, we compute the distance between a_k and the monitor of the path m_i , $d(m_i, a_k)$. This distance equals to the AS-level hop count from m_i to a_k along the path P_i if a_k is on the path P_i . Otherwise, $d(m_i, a_k)$ equals to the distance from m_i to a_k 's neighbor, who is both on P_i and the closest to m_i , plus 1. If a_k is not a member of a path neighborhood set $\mathcal{N}(P_i)$, the distance $d(m_i, a_k)$ is set to 0. The *total distance to*

Algorithm 2: The pseudo-code of locating algorithm

```

1 Initializing
2   set  $\mathcal{H}, \mathcal{C}, \mathcal{D}$  empty;
3 Updating
4   foreach  $m_i$  in the monitor set  $\mathcal{M}$ 
5     foreach  $a_k \in \mathcal{N}(P_i)$ 
6       if  $a_k \in \mathcal{H}$ 
7          $\mathcal{D}(a_k) += d(m_i, a_k)$ ;
8          $\mathcal{C}(a_k) += 1$ ;
9       else
10        insert  $a_k$  in  $\mathcal{H}$ ;
11         $\mathcal{C}(a_k) = 0$ ;
12         $\mathcal{D}(a_k) = d(m_i, a_k)$ ;
13 Ranking
14   sort  $a_k \in \mathcal{H}$  by  $\mathcal{C}(a_k)$ ;
15   for  $a_k$  with the same value of  $\mathcal{C}(a_k)$ ;
16     sort  $a_k$  by  $\mathcal{D}(a_k)$ ;

```

monitors equals to the summation of all $d(m_i, a_k)$.

After for each a_k in \mathcal{H} both *covered count* $\mathcal{C}(a_k)$ and *total distance to monitors* $\mathcal{D}(a_k)$ are computed, we rank all nodes in \mathcal{H} firstly based on their *covered count*. The greater the *covered count* a node a_k has, the higher it is ranked. Then for nodes having the same *covered count*, ties are broken by ranking them based on their *total distance to monitors*, –the lower the total distance, the higher the rank. If there are still ties, node ranks are determined randomly.

Hence, the final result of the locating algorithm is a list of nodes a_k , ordered based on how suspicious each node is being the hijacker. The most suspicious AS appears on the top of the list. The pseudo-code of the locating algorithm is shown in Algorithm 2.

The ranking algorithm described here may seem overly complicated for finding where the reported paths converge. However it is designed specifically to be robust against various measurement errors and possible hijacker countermeasures. One particular reason for this design is to reduce the effect of individual false paths. If a monitor-to-prefix path is changed due to reasons other than being hijacked and the monitor falsely assesses the situation as hijack, the path reported by this monitor may cause confusion on where the paths converge. Since it is difficult to distinguish this kind of paths beforehand, our algorithm has adopted the approach as described above to discredit the effect of these individual erroneous paths. For similar reasons, our ranking algorithm is robust against the IP-to-AS mapping errors if any.

Another reason for outputting an ordered list is that there are cases that hijacked paths converge before these paths reach the hijacker (*early converge*). This is more

likely to happen when the hijacker is located far away from the Internet core where the connectivity is rich. In this case the hijacked paths may converge at an upstream provider of the hijacker instead of the hijacker itself. Although as we will show later these hijacking scenarios typically have small impacts, in other words the portion of the Internet that is affected by such hijacks is small; still we wish to locate the hijacker. A list of suspects ranked by level of suspicion is well suited for addressing this issue.

5.3 Improvements

After the suspect list is computed, we can apply additional post-processing methods to further improve our results. The basic algorithm is very conservative in the way that \mathcal{H} includes all possible candidates. Now we look into ways that \mathcal{H} may be reduced. The hope is that with a trimmed suspect set to begin with, the locating algorithm can get more focused on the hijacker by increasing the rate that the most suspicious node on the list is the hijacker. Both improvements are designed to alleviate the early converge problem we mentioned before. Note that the improvements may exclude the real hijacker from the suspect set, but the evaluation (in Section 6.3.5) shows that chance is very small.

5.3.1 Improvement One: AS Relationship

In the basic algorithm, we have only taken AS topology into account. In other words, all topological neighbors of nodes on a reported AS path are added to the path's neighborhood set. In reality, not all physical connections between ASes are actively used for carrying traffic. In particular, some connections may be used only for traffic of one direction but not the other. This is largely due to profit-driven routing policies between different ISPs. Internet paths have been found to follow the "valley-free" property [10], i.e. after traversing a provider-to-customer edge or a peer edge, a path will not traverse another customer-to-provider path or another peer edge. If we constrain our suspect set using this AS relationship based property by removing the neighbors that do not follow the "Valley-free" property from the neighborhood set of each reported path, we are able to reduce the size of the neighborhood set and further on the suspect set \mathcal{H} .

One matter needs to be pointed out is that not all links on the reported paths are necessarily real due to the hijacker's countermeasures. Since we do not know what links are fabricated we should not trim the neighborhood sets too aggressively. We only perform this improvement on path links that we are reasonably certain that they are real. In particular, as we know that an attacker cannot forge path links that are before itself, thus we can rea-

sonably consider that on each reported path the links that are before the node immediately before the most suspicious node are real, and the trimming is only done on neighbors of these links.

This AS relationship based improvement is incorporated into the basic algorithm in an iterative fashion. We first pre-compute AS relationship information using method proposed in [10]. Note that this is done offline and does not require any real time access to the control plane information because AS relationship rarely change over time. After each execution of the basic algorithm produces a ranked suspect list, we can assume that on each path from the path's reporting monitor to the node immediately before the most suspicious node, all AS paths are valid. Based on these valid links, we can further infer the valid link in each neighborhood set. When there is any change of neighborhood set, we run the locating algorithm again to update the suspicious list. The iteration will stop if there is no change of suspicious list.

5.3.2 Improvement Two: Excluding Innocent ASes

The second improvement focuses on removing nodes from the suspect set \mathcal{H} of whose innocence we are reasonably certain. One group of these nodes are the ones that are on the reported paths that actually pass through the most suspicious node and before the most suspicious node. The reason for this exclusion is again that the attacker cannot forge the identity of these nodes.

The second group of the innocent nodes are selected based on the path disagreement test described in [45]. In path disagreement test, a reference point that is outside of the target prefix but topologically very close to the prefix is selected and the path from a monitor to this reference point and the path from same monitor to the target prefix are compared. If they differ significantly it is highly likely that the prefix has been hijacked. The high accuracy of this test leads us to believe that nodes on monitor-to-reference point paths are not likely to be the hijacker. They can be excluded from the suspect set.

The second improvement is again incorporated into the basic algorithm in an iterative fashion. After each execution of the basic algorithm, the suspect set is reduced by removing nodes of the two aforementioned innocent groups. Then basic algorithm is executed again using the reduced suspect set. The iteration is repeated until the basic suspect set is stable.

6 Evaluation

We implemented and deployed LOCK on PlanetLab [31]. This is a necessary step to show that LOCK is deployable in real world system. Also using the PlanetLab testbed, we evaluated the performance of LOCK

based on measurements of the deployed LOCK system. In this section, we first present our measurement setup and evaluation methodology. Then we evaluate the performance of the monitor selection algorithm in LOCK, and the effectiveness of LOCK against synthetic hijacks, reconstructed previously-known hijacking events, and real hijacking attacks launched by us.

6.1 Measurement Setup

6.1.1 Candidate Monitors

In our experiments, we first chose a number of geographically diversified PlanetLab [31] nodes as candidate network location monitors. We manually selected 73 PlanetLab nodes in 36 distinct ASes in different geographical regions. More specifically, relying on their DNS names, half of the nodes are in the U.S., covering both coasts and the middle. The other half were selected from other countries across multiple continents. Among these candidate monitors, a set of monitors were selected using the algorithm presented in Section 4 to monitor each target prefix.

6.1.2 Target Prefixes

We selected target prefixes from four different sources: (i) Multiple Origin ASes (MOAS) prefixes, (ii) Single Origin AS (SOAS) prefixes with large traffic volume, (iii) prefixes of popular Web sites, and (vi) prefixes of popular online social networks. To get prefixes from sources (i) and (ii), we first use BGP tables obtained from RouteViews [3] and RIPE [2] to identify the initial candidates of MOAS and SOAS prefixes. Then for each candidate prefix, we tried to identify a small number (up to 4) of live (*i.e.* responsive to *ping*) IP addresses. To avoid scanning the entire candidate prefixes for live IP addresses, we mainly used the prefixes' local DNS server IP addresses to represent the prefix. If we failed to verify any live IP address for a particular prefix, we discarded this prefix from our experiments. Using this method, we selected 253 MOAS prefixes. We also ranked all SOAS prefixes based on "popularity" (*i.e.* traffic volume observed at a Tier-1 ISP based on Netflow) of the prefix and selected top 200 prefixes with live local DNS server IP addresses.

We also selected prefixes that correspond to popular applications on the Internet: Web and online social networks. In particular, we selected the top 100 popular Web sites based on the Alex [5] ranking and obtain their IP addresses and corresponding prefixes. We also obtained IP addresses and prefixes of YouTube and 50 popular online social networks. Each of the selected online social networks has at least 1 million registered users in

multiple countries. Combining prefixes from all above four sources, we have a total of 451 target prefixes.

6.1.3 Measurement Data Gathering

In our experiments, each monitor measures its paths to all selected IP addresses in all target prefixes via *traceroute*. We also measured paths from each monitor to reference points of target prefixes [45]. In addition, each monitor also measures its paths to other monitors. We obtain AS-level paths of above measured paths by mapping IP addresses to their ASes based on the IP-to-AS mapping published at iPlane [15].

The results presented here are based on monitoring data collected from March 20th, 2008 to April 20th, 2008. In particular, we measured each path (from a monitor to a target prefix) every 5 minutes.

In addition, we obtained the AS topology data during the same time period from [16]. We also used the AS relationship information captured for customer-to-provider and peer links over 6 month (from June 2007 to December 2007) using the inferring technique described in [24].

6.2 Evaluation Methodology

We evaluated LOCK based on three sets of prefix hijacking experiments: (i) synthetic prefix hijacking events based on Internet measurement data; (ii) reconstructed previously-known prefix hijacking events based on Internet measurement data; and (iii) prefix hijacking events launched by us on the Internet.

6.2.1 Simulating Synthetic Prefix Hijacking Events

We consider commonly used interdomain routing policies: "prefer customer routes" and "valley-free routing". In particular, an AS prefers routes announced from its customer ASes over those announced from its peer ASes, further over those announced from its provider ASes. These policies are driven by financial profit of ASes. If two routes have the same profit-based preference, then the shorter route (*i.e.*, fewer AS hop count) is preferred. When the hijacker announces a fake prefix, we assume that it does this to all its neighbors (*i.e.* providers, peers, and customers) to maximize hijacking impact.

For each attack scenario, we simulated all three types of hijacking scenarios, namely imposture, interception, malicious, as shown in Figure 2 in Section 3. Each attack scenario is simulated as follows. In each attack scenario, we selected one PlanetLab node as the hijacker h and another PlanetLab node as the target prefix t . The hijacking is then observed from the monitors.

In the imposture scenario, the path from s to t will become the path from s to h if s is polluted by h 's attack. Otherwise, the path from s to t remains the same as

before the attack. This was repeated for all possible selections of h , t , and s , except for cases where t 's AS is on the AS path from s to h because the hijack will never succeed in these cases. In addition, since some paths were not traceroute-able, we had to discard combinations that require these paths.

The setup for simulating interceptions and malicious scenarios is similar to that of the imposture scenario. In the interception scenario, the path from s to t will be the concatenation of paths from s to h and from h to t if s is polluted by h 's attack. However, we exclude the cases that there is one or more common ASes between these two paths. This is because the hijacker h cannot successfully redirect the traffic back to the target prefix t , i.e., the interception hijack fails.

In the malicious scenario, the hijacker h has countermeasure against LOCK. The path from s to t will be the path from s to h (the AS of h will not show up) with a few random AS hops appended after h . The generation of these random AS hops is kind of tricky. If h generates different noisy tails for different monitors, these tails may not converge at all. In this case, it is easier for our locating algorithm to locate the hijacker. In our simulations, in anticipating that the hijacker may fill its replies to traceroute probes with fake identities, we replaced the node identities with random entries for all nodes that are farther than the hijacker (inclusive) in the paths resulted from running traceroute from different monitors.

6.2.2 Reconstructing Previously-Known Prefix Hijacking Events

We obtained the list of previously-known prefix hijacking events from the Internet Alert Registry [14]. IAR provides the network operator community with the up to date BGP (Border Gateway Protocol) routing security information. Its discussion forum³ posts suspicious hijacking events. We chose 7 that had been verified and confirmed to be prefix hijacking events, including some famous victims such as YouTube and eBay, during a time period from 2006 to 2008.

We reconstructed these 7 hijacking events using the following method. First, we selected a traceroutable IP in each victim AS as the probing target t , and a traceroutable IP in each hijacker AS as the hijacker h . Then we collected the traceroute information from each monitoring site s to these targets t and hijackers h . The routing policy is based again on the profit driven model. Since we don't know what kind of behavior each hijacker took (imposture, interception or malicious), We conservatively assume that the hijacker will try to evade our measurement. So it follows the malicious scenario we mentioned before.

6.2.3 Launching Controlled Prefix Hijacking Events

We conducted controlled prefix hijacking experiments on the Internet using hosts under our control at four different sites, namely Cornell, Berkeley, Seattle, and Pittsburgh. Each host ran the Quagga software router and established eBGP sessions with different ISPs. Effectively, this allowed us to advertise our dedicated prefix (204.9.168.0/22) into the Internet through the BGP sessions. The idea behind the experiments was to use our prefix as the target prefix with one of the sites serving as the owner of the prefix and the other three sites (separately) serving as the geographically distributed attackers trying to hijack the prefix. More implementation details can be found in [6]. In our experiment, we focused on the imposture scenario. There were 12 hijacking cases by switching the role of each site. These attacks were launched according to a pre-configured schedule during period from May 2, 2008 to May 4, 2008.

6.2.4 Performance Metrics

LOCK identifies suspicious hijackers and ranks them based on their likelihood of being the true hijacker. The hijacker ranked at top one is most suspicious. We thus define the *top- n accuracy* of LOCK as the percentage of hijacking events that the true hijacker ranks as top n on the suspect list, where n is a parameter. We use this parameterized definition because different operators might have different preference. Some might prefer knowing just the most suspicious hijacker, in which top-1 accuracy is most important. Others might not mind learning a longer suspect list to increase the likelihood that the hijacker is included in the suspect list. We will later show that the top-2 accuracy is already very high.

In addition, we define *impact* of a hijacker h as the fraction of the ASes from which the traffic to the target prefix t is hijacked to h , similar to what is done in [23]. We will then study the correlation between LOCK's locating accuracy of a given hijacker and the impact of its attack.

6.3 Evaluation on Synthetic Prefix Hijacking Events

In this section, we use the results of LOCK based on the data plane measurement to illustrate our findings.

6.3.1 Monitor Selection

We compare the performance of the monitor selection algorithm (referred as *clustering and ranking*) proposed in Section 4 with the following three monitor selection algorithms: (i) *random*: randomly selecting m monitors

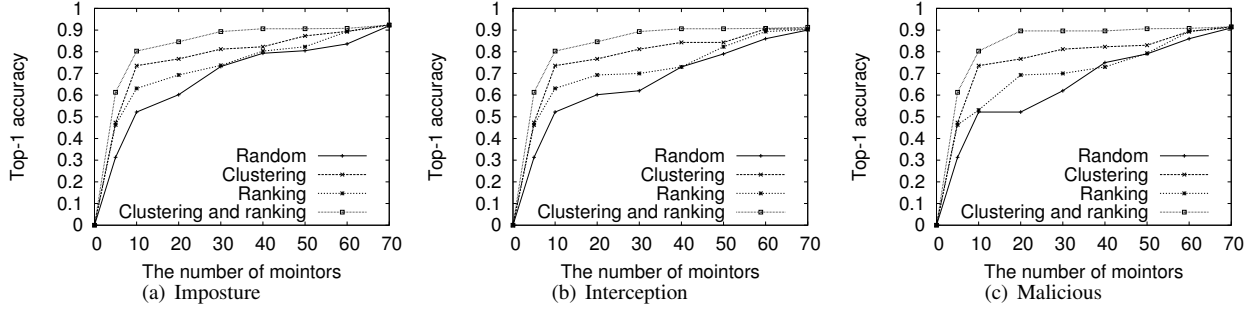


Figure 3: Performance of monitor selection algorithms

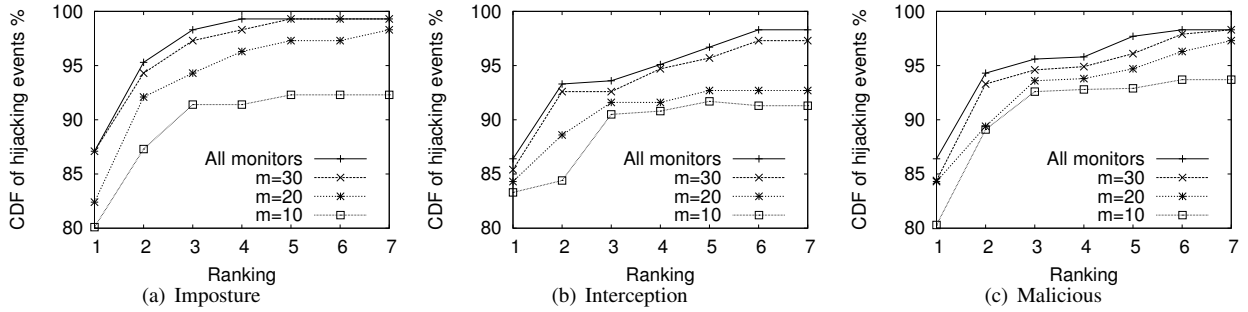


Figure 4: The CDF of the rank of hijackers in synthetic attacks

from all M candidates. (ii) *clustering*: dividing M monitors into clusters based on the clustering algorithm proposed in Section 4.1, then randomly selecting one monitor from each cluster; and (iii) *ranking*: ranking M monitors based on the ranking algorithm proposed in Section 4.2, then selecting the first m candidates.

Figure 3 shows the top-1 accuracy of different monitor selection algorithms when varying the subsets of monitors. We focused on synthetic attacks since the dataset is much larger than previously-known hijacks and controlled real hijacks. We find that: (i) There is always a trade-off between the number of monitors selected and hijacker-locating accuracy. Note that even using all 73 monitors, the accuracy is less than 92%. It is not surprising because it is hard to detect the hijacking events which have small impact [23]. (ii) The *clustering and ranking* algorithm outperforms the rest. For example, for imposture attacks, selecting 10 monitors based on the ranking and clustering algorithm is enough for achieving 80% top-1 accuracy. This is only 1/3 of number of monitors needed to reach the same top-1 accuracy with either *ranking* or the *clustering* algorithm, or 1/6 if monitors are selected randomly. Hence in our experiments in the rest of the section, whenever we need to select monitors, we use the *clustering and ranking* algorithm, unless otherwise specified.

Moreover, we want to make sure that the monitor se-

lection algorithm does not overload any monitors by assigning too many target prefixes to it for monitoring. For each target prefix we select $m = 30$ monitors from the total pool of $M = 73$ candidate monitors using the monitor selection algorithm described in Section 4. Individual monitor's work load is computed as the number of target prefixes assigned to it divided by the total number of target prefixes. Ideally, the average work load, which is the load each monitor gets if the monitoring tasks are evenly across all monitors equally instead of assigning prefixes to monitors that can monitor most effectively, is $m/M \approx 0.4$. As a comparison, we observe the real workload ranges from 0.3 to 0.55. In addition, only 4 monitors out of 73 have load above 0.5, which means that they monitor more than half of prefix targets.

6.3.2 Effectiveness of Basic Algorithm

The evaluations of two different aspects of the effectiveness of the hijacker-locating algorithm are presented in this section. We show how well the ranked list captures the hijacker identity, as well as how well the ranked list reflects the impact of the hijack events.

Figure 4 illustrates where the hijacker is ranked in the suspect list produced by the basic algorithm, for different number of monitors selected. Obviously, the higher the hijacker is ranked, the better the basic algorithm is. From this figure, we can see that:

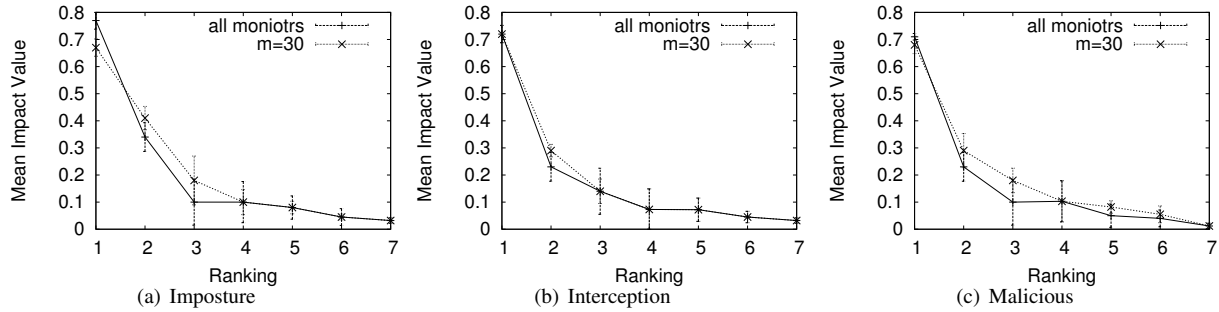


Figure 5: Correlating the impact with the ranking value

- More than 80% of the time, our basic algorithm pinpoints the hijacker by ranking it as *top 1* on the suspect list, regardless what kind of attack and with how many monitors, as long as more than the minimum number of 10 monitors.
- Because of the early convergence problem described in Section 5.2, the hijacker may not be ranked the first. Therefore as we look into not only the highest ranked node but even more nodes, the chance that the hijacker is included in this selective set increases. For example with 10 monitors, the chance that an imposture hijacker is found among the top three nodes is more than 94%, a 14% increase from only looking at the highest ranked suspect.
- The hijacker-locating algorithm performs best in imposture scenarios. The reason is that imposture paths are more likely to be straight without detouring.
- Obviously the more monitors we employ, the better the algorithm works. What is interesting is that seemingly by having $m = 30$ we have reached the point of diminishing return: having more than 30 monitors no longer improves the performance much.

Next, we study the relationship between the impact of a hijack event and where the hijacker is ranked in the suspect list. This shows another aspect of the quality of our hijacker-locating algorithm. That is, not only we want to locate hijackers, we especially want to locate the hijackers causing great damages. Figure 5 shows the ranking (x-axis) vs the median impact of all hijackers with the same ranking (Y-axis). All three plots in Figure 5 show that there is a positive relationship between the hijacker's rank and the impact of its hijack attack. In other words, the larger the impact caused by a hijacker, the more likely our locating algorithm will rank the hijacker high in the suspect list. This is mostly due to the fact that the early

converge problems occur mostly at where hijacks have small impacts, near Internet edge.

6.3.3 Effectiveness of Improvements

Finally, we evaluate the quality of two improvements (I1 and I2) proposed in Section 5.3. In particular, we are not only interested in the increase in top-1 accuracy these improvements may bring, but also the false negative rate (FNR), which is the ratio that the improvements mistakenly exclude a hijacker from the suspect list.

Table 1 shows both sets of numbers for different kinds of attacks and different number of monitors. Different combinations of the basic algorithm and the improvements are shown in different rows of the table.

- I2 helps more. The reason is that for I1 we can only trust the path before converges. But for I2, we have more information provided by the reference point traceroute.
- When combining I1 and I2, the accuracy can be further improved. This is because the nodes that I1 and I2 remove from the suspect list are typically not the same.
- In general, LOCK (i.e., B+I1+I2) is able to pinpoint the prefix hijacker AS with an accuracy of over 91%, up to 94.3%.
- The false negative ratio introduced by improvements is relatively low. For example, when using all monitors we can improve the accuracy by more than 5% by applying both I1 and I2, while the false negative ratio resulted from applying the improvements is only 0.09%

6.3.4 Effectiveness on different AS-levels

We study the locating accuracy when the hijacker located in different level in the AS hierarchy. We classify AS nodes into three tiers: Tier-1 nodes, transit nodes, and

Table 1: The effectiveness of improvement

Algorithms	All monitors						m=30					
	Imposture		Interception		Malicious		Imposture		Interception		Malicious	
	Accuracy	FNR	Accuracy	FNR	Accuracy	FNR	Accuracy	FNR	Accuracy	FNR	Accuracy	FNR
B	88.7%	0.00%	86.3%	0.00%	85.4%	0.00%	86.2%	0.00%	84.7%	0.00%	83.5%	0.00%
B+I1	89.8%	0.03%	90.3%	0.17%	88.6%	0.14%	86.4%	0.05%	85.3%	0.14%	84.6%	0.11%
B+I2	91.3%	0.09%	93.1%	0.16%	90.4%	0.10%	90.7%	0.14%	90.6%	0.18%	88.3%	0.20%
B+I1+I2	94.2%	0.09%	94.3%	0.24%	93.1%	0.18%	92.4%	0.20%	91.4%	0.17%	91.8%	0.26%

Table 2: The effectiveness on different AS-levels

Category	Imposture		Interception		Malicious	
	Accuracy	FNR	Accuracy	FNR	Accuracy	FNR
All	92.4%	0.20%	91.4%	0.17%	91.8%	0.26%
Transit	97.6%	0.04%	96.3%	0.07%	94.8%	0.14%
Stub	90.2%	0.18%	90.1%	0.21%	90.4%	0.35%

Table 3: The effectiveness on prevention after locating

Methods	Initial	Stop the origin	Stop in Tier1
LOCK	23.43%	0.10%	2.31%
Simple Locating	23.43%	13.13%	21.90%

stub nodes like in [23].⁴ Our hijackers in planetlab belongs to transit nodes, or stub nodes. When using two improvements and 30 monitors, we compare the accuracy and false negative ration for these two classes, in Table 2. The hijackers on the higher level could be located more easily. The hijackers on the edge is relatively hard to locate. We can still achieve more than 90% accuracy.

6.3.5 Effectiveness of filtering after locating the hijacker

After locating the AS, the next step is to filter the fake AS announcement from it. We compare the average percentage of impacted (polluted) AS, before and after the locating and filtering either stop on the origin or on the Tier1 AS. As a comparison, we also select the last hop of AS of the observed paths as a hijacker (simple locating approach) then do the same filtering. They are under malicious case. Table 3 shows that Lock is more helpful than simple locating method to prevent hijacks.

6.3.6 Remarks

We have shown that LOCK performs well using monitor-to-prefix paths measured in the data plane. Similar observation would hold if control plane paths are used in LOCK. In the non-malicious cases, the monitor-to-prefix paths that observed in the control plane are the same as those observed in the data plane. However, in the malicious case, we have shown that the hijacker can employ more sophisticated evasion technique in the data plane than in the control plane. Therefore, our results shown

Table 4: Previously-Known prefix hijacking events

Victim AS	Hijacker AS	Date	#monitors
3691	6461	March 15, 2008	16
36561 (YouTube)	17557	February 24, 2008	9
11643 (eBay)	10139	November 30, 2007	7
4678	17606	January 15, 2007	8
7018	31604	January 13, 2007	13
1299	9930	September 7, 2006	5
701, 1239	23520	June 7, 2006	12

in this section provide a lower bound of LOCK performance against malicious hijackers.

6.4 Evaluation on Previous-Known Attacks

We reconstructed 7 previously known prefix hijacking events. Table 4 shows the dates and ASes of the hijacker and the target prefix (i.e., the victim) of these events. By using all 73 monitors deployed on PlanetLab, LOCK is able to accurately locate the hijacker ASes as the top-1 suspects for all these hijacking events, i.e., the true hijackers are ranked first on the suspect lists. Using the monitor selection algorithm (clustering and ranking) presented in Section 4, we also identified the minimum set of monitors that were required by LOCK to accurately locate the hijacker in each of these previously-known events. The last column of Table 4 shows that all hijackers could be correctly located as top-1 suspects by using 16 or fewer monitors. A detailed investigation shows that these hijacks polluted majority of the monitors, resulting in LOCK's high locating accuracy.

6.5 Evaluation on Controlled Real Attacks

In this set of experiments, we launched real imposture attacks using four sites under our control. The schedule is shown in Table 5. During the experiments each LOCK

Table 5: Locating hijackers in real Internet attacks

Victim Site	Hijacker Site	Launch Time (EST)	Response Time (minutes)	Required monitors
Cornell	Berkeley	May 2 12:01:31	13	12
	Seattle	May 2 16:12:47	7	10
	Pittsburgh	May 2 17:34:39	9	9
Pittsburgh	Cornell	May 2 19:32:09	13	14
	Berkeley	May 2 22:50:25	11	15
	Seattle	May 3 02:26:26	12	15
Seattle	Cornell	May 3 11:20:42	9	8
	Pittsburgh	May 3 13:03:10	12	12
	Berkeley	May 3 19:16:16	8	18
Berkeley	Seattle	May 3 22:35:07	13	14
	Pittsburgh	May 4 00:01:01	12	16
	Cornell	May 4 11:19:20	11	10

monitor probed the target prefix 204.9.168.0/22 once every 5 minutes. For the purpose of this experiment, we used the detection scheme proposed in [45], which was able to detect all the attacks launched from the controlled sites. The hijackers in these experiments were “honest”, i.e., no countermeasure was done by the hijackers. Thus we observed that LOCK locates the hijackers as top-1 suspects in all the real imposture attacks.

In this real Internet experiment, we were able to evaluate the response time of LOCK in addition to its accuracy. The response time is defined as the latency from the time the attack is launched by the hijacker to the time that LOCK locates the hijacker. The response time highly depends on two major factors: the speed of propagation of invalid route advertisement and the probing rate employed by LOCK monitors. It usually takes up to a few minutes for a route advertisement to spread across the Internet. This is the latency that an attack takes before making full impact on the Internet. After a LOCK monitor is impacted by an attack, it may also take a few minutes for the monitor to detect and locate the hijacker because the monitor probes target prefixes periodically. There are also few minor factors that may affect the response time. For example, there can be a few seconds latency for LOCK monitors to get replies for each probe. However, they are neglected in our evaluation because they are orders of magnitude smaller than the above two major factors.

We record the timestamp each attack is launched from a control site and the timestamp LOCK locates the hijacker (i.e., that controlled site). Both of which are synchronized with a common reference time server. The response time is computed by taking the difference between the above two timestamps. If alternative detection scheme is used, the observed response time serves as a conservative upper bound of the latency that LOCK takes to locate the hijacker.

Table 5 shows the response time and minimum number of required monitors for locating these real prefix hijack-

ing events. We observe that LOCK is able to locate the hijacker within 7 ~ 13 minutes. Given that the probe frequency of LOCK monitors is 5 minutes, the results implies that it takes LOCK at most 2 ~ 3 rounds of probes to detect and locate the hijacker. Moreover, all hijackers are correctly located as top-1 suspects by using 18 or fewer monitors.

7 Related Work

A number of solutions have been proposed to proactively defend against prefix hijacking. They can be categorized into two broad categories: crypto based and non-crypto based. Crypto based solutions, such as [4, 8, 13, 19, 27, 35, 36], require BGP routers to sign and verify the origin AS and/or the AS path to detect and reject false routing messages. However, such solutions often require signature generation and verification which have significant impact on router performance. Non-crypto based proposals such as [11, 18, 32, 37, 44] require changing router softwares so that inter-AS queries are supported [11, 32], stable paths are more preferred [18, 37], or additional attributes are added into BGP updates to facilitate detection [44]. All the above proposals are not easily deployable because they all require changes in router software, router configuration, or network operations, and some also require public key infrastructures.

Recently, there has been increasing interest in solutions for reactive detection of prefix hijacking [6, 12, 21, 22, 26, 34, 36, 45] because such solutions use passive monitoring and thus are highly deployable. For example, [43, 45] monitor the data plane, [21, 22, 26, 34] monitor the control plane, and [6, 12, 36] monitor both control and data planes. LOCK is different from all these approaches because LOCK locates the hijacker AS for each prefix hijacking event, while the above approaches only focus on detecting a hijacking event without further revealing the location of the hijacker. In fact, LOCK can be used together with any of the above hijacking detec-

tion algorithm for identifying hijacker AS because the flexibility of LOCK on using either control plane or data plane information in locating hijacker.

Measurement-based solutions often require careful selection of monitors. In particular, LOCK selects monitors based on their likelihood of observing hijacking events, while [45] proposed an initial monitor selection algorithm to detect hijacks without further evaluation, and [23] tries to understand the impact of hijackers in different locations. In addition, there have been a number of studies [7,9,40] on the limitations of existing BGP monitoring systems (e.g. RouteView) and the impacts of monitor placement algorithms [29] for collecting BGP data for a boarder range of applications such as topology discovery, dynamic routing behavior discovery and network black hole discovery [20,41].

Finally, existing works [38,39,42] proposed to mitigating prefix hijacking by using an alternative routing path [38,39], or by modifying AS_SET [42]. Though LOCK does not directly handle the mitigation of prefix hijacking events, LOCK can provide the hijacker location information required by these mitigation schemes.

8 Conclusion

In this paper, we propose a robust scheme named LOCK for locating the prefix hijacker ASes based on distributed AS path measurements. LOCK has several advantages: 1) LOCK is an unified scheme that locates hijackers in the same fashion across different types of prefix hijacking attacks; 2) LOCK is a distributed scheme with workload distributed among multiple monitors; 3) LOCK is a robust scheme because multiple monitors help improving locating accuracy and discounting individual errors; and 4) LOCK is a flexible scheme because it can use AS path measurement data obtained either from data-plane or from control-plane to locate the hijacker AS.

The performance of the LOCK scheme has been evaluated extensively through experiments in three kinds of settings: test topology constructed based on real Internet measurements, reconstructed known prefix hijack attacks, and controlled prefix hijack attacks conducted on the Internet. We have shown that the LOCK scheme is very accurate, highly effective, and rapid reacting.

Acknowledgement

Tongqing Qiu and Jun Xu are supported in part by NSF grants CNS-0519745, CNS-0626979, CNS-0716423, and CAREER Award ANI-023831.

References

- [1] <http://www.ripe.net/news/study-youtube-hijacking.html>.

- [2] RIPE RIS Raw Data. <http://www.ripe.net/projects/ris/rawdata.html>.
- [3] University of Oregon Route Views Archive Project. <http://www.routeview.org>.
- [4] AIELLO, W., IOANNIDIS, J., AND MCDANIEL, P. Origin Authentication in Interdomain Routing. In *Proc. of ACM CCS* (Oct. 2003).
- [5] Alexa. <http://www.alexa.com/>.
- [6] BALLANI, H., FRANCIS, P., AND ZHANG, X. A Study of Prefix Hijacking and Interception in the Internet. In *Proc. ACM SIGCOMM* (Aug. 2007).
- [7] BARFORD, P., BESTAVROS, A., BYERS, J., AND CROVELLA, M. On the marginal utility of network topology measurements. In *IMW '01* (New York, NY, USA, 2001), ACM, pp. 5–17.
- [8] BUTLER, K., MCDANIEL, P., AND AIELLO, W. Optimizing BGP Security by Exploiting Path Stability. In *Proc. ACM CCS* (Nov. 2006).
- [9] COHEN, R., AND RAZ, D. The Internet Dark Matter - on the Missing Links in the AS Connectivity Map. In *INFOCOM* (2006).
- [10] GAO, L. On Inferring Autonomous System Relationships in the Internet. *IEEE/ACM Transactions on Networking* (2001).
- [11] GOODELL, G., AIELLO, W., GRIFFIN, T., IOANNIDIS, J., MCDANIEL, P., AND RUBIN, A. Working Around BGP: An Incremental Approach to Improving Security and Accuracy of Interdomain Routing. In *Proc. NDSS* (Feb. 2003).
- [12] HU, X., AND MAO, Z. M. Accurate Real-time Identification of IP Prefix Hijacking. In *Proc. IEEE Security and Privacy* (May 2007).
- [13] HU, Y.-C., PERRIG, A., AND SIRBU, M. SPV: Secure Path Vector Routing for Securing BGP. In *Proc. ACM SIGCOMM* (Aug. 2004).
- [14] IAR. <http://iar.cs.unm.edu/>.
- [15] iPlane. <http://iplane.cs.washington.edu/>.
- [16] Internet topology collection. <http://irl.cs.ucla.edu/topology/>.
- [17] JOHNSON, S. Hierarchical Clustering Schemes. In *Psychometrika* (1967).
- [18] KARLIN, J., FORREST, S., AND REXFORD, J. Pretty Good BGP: Protecting BGP by Cautiously Selecting Routes. In *Proc. IEEE ICNP* (Nov. 2006).
- [19] KENT, S., LYNN, C., AND SEO, K. Secure Border Gateway Protocol (S-BGP). *IEEE JSAC Special Issue on Network Security* (Apr. 2000).
- [20] KOMPELLA, R. R., YATES, J., GREENBERG, A., AND SNOEREN, A. C. Detection and Localization of Network Black Holes. In *Proc. IEEE INFOCOM* (2007).
- [21] KRUEGEL, C., MUTZ, D., ROBERTSON, W., AND VALEUR, F. Topology-based Detection of Anomalous BGP Messages. In *Proc. RAID* (Sept. 2003).
- [22] LAD, M., MASSEY, D., PEI, D., WU, Y., ZHANG, B., AND ZHANG, L. PHAS: A Prefix Hijack Alert System. In *Proc. USENIX Security Symposium* (Aug. 2006).
- [23] LAD, M., OLIVEIRA, R., ZHANG, B., AND ZHANG, L. Understanding Resiliency of Internet Topology Against Prefix Hijack Attacks. In *Proc. IEEE/IFIP DSN* (June 2007).
- [24] MAO, Z. M., QIU, L., WANG, J., AND ZHANG, Y. On AS-Level Path Inference. In *Proc. ACM SIGMETRICS* (2005).

- [25] MAO, Z. M., REXFORD, J., WANG, J., AND KATZ, R. Towards an Accurate AS-level Traceroute Tool. In *Proc. ACM SIGCOMM* (2003).
- [26] RIPE myASn System. <http://www.ris.ripe.net/myasn.html>.
- [27] NG, J. Extensions to BGP to Support Secure Origin BGP. <ftp://ftp-eng.cisco.com/sobgp/drafts/draft-ng-sobgp-bgp-extensions-02.txt>, April 2004.
- [28] NORDSTROM, O., AND DOVROLIS, C. Beware of BGP Attacks. *ACM SIGCOMM Computer Communications Review (CCR)* (Apr. 2004).
- [29] OLIVEIRA, R., LAD, M., ZHANG, B., PEI, D., MASSEY, D., AND ZHANG, L. Placing BGP Monitors in the Internet. UW Technical Report, 2006.
- [30] OLIVEIRA, R., PEI, D., WILLINGER, W., ZHANG, B., AND ZHANG, L. In Search of the elusive Ground Truth: The Internet's AS-level Connectivity Structure. In *Proc. ACM SIGMETRICS* (2008).
- [31] PlanetLab. <http://www.planet-lab.org>.
- [32] QIU, S. Y., MONROSE, F., TERZIS, A., AND MCDANIEL, P. D. Efficient Techniques for Detecting False Origin Advertisements in Inter-domain Routing. In *Proc. IEEE NPsec* (Nov. 2006).
- [33] RAMACHANDRAN, A., AND FEAMSTER, N. Understanding the Network-Level Behavior of Spammers. In *Proceedings of ACM SIGCOMM* (2006).
- [34] SIGANOS, G., AND FALOUTSOS, M. Neighborhood Watch for Internet Routing: Can We Improve the Robustness of Internet Routing Today? In *Proc. IEEE INFOCOM* (May 2007).
- [35] SMITH, B. R., AND GARCIA-LUNA-ACEVES, J. J. Securing the Border Gateway Routing Protocol. In *Proc. Global Internet* (Nov. 1996).
- [36] SUBRAMANIAN, L., ROTH, V., STOICA, I., SHENKER, S., AND KATZ, R. H. Listen and Whisper: Security Mechanisms for BGP. In *Proc. USENIX NSDI* (Mar. 2004).
- [37] WANG, L., ZHAO, X., PEI, D., BUSH, R., MASSEY, D., MANKIN, A., WU, S., AND ZHANG, L. Protecting BGP Routes to Top Level DNS Servers. In *Proc. IEEE ICDCS* (2003).
- [38] XU, W., AND REXFORD, J. Don't Secure Routing Protocols, Secure Data Delivery. In *Proc. ACM HotNets* (2006).
- [39] XU, W., AND REXFORD, J. MIRO: multi-path interdomain routing. In *Proc. ACM SIGCOMM* (2006).
- [40] ZHANG, B., LIU, R. A., MASSEY, D., AND ZHANG, L. Collecting the Internet AS-level Topology. *Computer Communication Review* 35, 1 (2004), 53–61.
- [41] ZHANG, Y., ZHANG, Z., MAO, Z. M., HU, Y. C., , AND MAGGS, B. On the Impact of Route Monitor Selection. In *Proceedings of ACM IMC* (2007).
- [42] ZHANG, Z., YANG, Y., HU, Y. C., AND MAO, Z. M. Practical Defenses Against BGP Prefix Hijacking. In *Proc. of CoNext* (Dec. 2007).
- [43] ZHANG, Z., ZHANG, Y., HU, Y., MAO, Z., AND BUSH, R. iSPY: Detecting IP Prefix Hijacking on My Own. In *Proc. ACM SIGCOMM* (Aug. 2008).
- [44] ZHAO, X., PEI, D., WANG, L., MASSEY, D., MANKIN, A., WU, S., AND ZHANG, L. Dection of Invalid Routing Announcement in the Internet. In *Proc. IEEE/IFIP DSN* (June 2002).
- [45] ZHENG, C., JI, L., PEI, D., WANG, J., AND FRANCIS, P. A Light-Weight Distributed Scheme for Detecting IP Prefix Hijacks in Real-Time. In *Proc. ACM SIGCOMM* (Aug. 2007).

Notes

¹Note that some vendor implementation does not check whether the neighbor has appended its own AS in the announcement , while some vendor implementation does check (in which this hijack does not succeed).

²The complexity is not a concern here because the number of clusters is relatively small comparing to traditional clustering problem.

³Discussion form: <http://iar.cs.unm.edu/phpBB2/viewforum.php?f=2>

⁴To choose the set of Tier-1 nodes, we started with a well known list, and added a few high degree nodes that form a clique with the existing set. Nodes other than Tier-1s but provide transit service to other AS nodes, are classified as transit nodes, and the remainder of nodes are classified as stub nodes.

GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code

Salvatore Guarnieri
University of Washington
sammyg@cs.washington.edu

Benjamin Livshits
Microsoft Research
livshits@microsoft.com

Abstract

The advent of Web 2.0 has led to the proliferation of client-side code that is typically written in JavaScript. This code is often combined — or mashed-up — with other code and content from disparate, mutually untrusting parties, leading to undesirable security and reliability consequences.

This paper proposes GATEKEEPER, a mostly static approach for soundly enforcing security and reliability policies for JavaScript programs. GATEKEEPER is a highly extensible system with a rich, expressive policy language, allowing the hosting site administrator to formulate their policies as succinct Datalog queries.

The primary application of GATEKEEPER this paper explores is in reasoning about JavaScript widgets such as those hosted by widget portals Live.com and Google/IG. Widgets submitted to these sites can be either malicious or just buggy and poorly written, and the hosting site has the authority to reject the submission of widgets that do not meet the site's security policies.

To show the practicality of our approach, we describe nine representative security and reliability policies. Statically checking these policies results in 1,341 verified warnings in 684 widgets, no false negatives, due to the soundness of our analysis, and false positives affecting only two widgets.

1 Introduction

JavaScript is increasingly becoming the lingua franca of the Web, used both for large monolithic applications and small *widgets* that are typically combined with other code from mutually untrusting parties. At the same time, many programming language purists consider JavaScript to be an atrocious language, forever spoiled by hard-to-analyze dynamic constructs such as `eval` and the lack of static typing. This perception has led to a situation where code instrumentation and not static program analysis has been the weapon of choice when it comes to enforcing security

policies of JavaScript code [20, 25, 29, 35].

As a recent report from Finjan Security shows, widget-based attacks are on the rise [17], making widget security an increasingly important problem to address. The report also describes well-publicised vulnerabilities in the Vista sidebar, Live.com, and Yahoo! widgets. The primary focus of this paper is on statically enforcing security and reliability policies for JavaScript code. These policies include restricting widget capabilities, making sure built-in objects are not modified, preventing code injection attempts, redirect and cross-site scripting detection, preventing global namespace pollution, taint checking, etc. Soundly enforcing security policies is harder than one might think at first. For instance, if we want to ensure a widget cannot call `document.write` because this construct allows arbitrary code injection, we need to either analyze or disallow tricky constructs like `eval("document" + ".write('...')")`, or `var a = document['wri' + 'te']; a('...')`; which use reflection or even

```
var a = document;
var b = a.write;
b.call(this, '...')
```

which uses aliasing to confuse a potential enforcement tool. A naïve unsound analysis can easily miss these constructs. Given the availability of JavaScript obfuscators [19], a malicious widget may easily masquerade its intent. Even for this very simple policy, `grep` is far from an adequate solution.

JavaScript relies on heap-based allocation for the objects it creates. Because of the problem of object aliasing alluded to above in the `document.write` example where multiple variable names refer to the same heap object, to be able to soundly enforce the policies mentioned above, GATEKEEPER needs to statically reason about the program heap. To this end, this paper proposes the first points-to analysis for JavaScript. The programming language community has long recognized pointer analysis to be a key building block for reasoning about object-oriented programs. As a result, pointer analy-

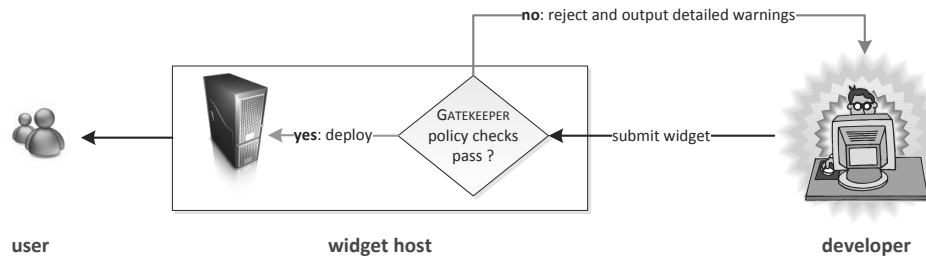


Figure 1: GATEKEEPER deployment. The three principals are: the *user*, the *widget host*, and the *widget developer*.

ses have been developed for commonly used languages such as C and Java, but nothing has been proposed for JavaScript thus far. However, a *sound* and precise points-to analysis of the *full* JavaScript language is very hard to construct. Therefore, we propose a pointer analysis for JavaScript_{SAFE}, a realistic subset that includes prototypes and reflective language constructs. To handle programs outside of the JavaScript_{SAFE} subset, GATEKEEPER inserts runtime checks to preclude dynamic code introduction. Both the pointer analysis and nine policies we formulate on top of the points-to results are written on top of the same expressive Datalog-based declarative analysis framework. As a consequence, the hosting site interested in enforcing a security policy can program their policy in several lines of Datalog and apply it to all newly submitted widgets.

In this paper we demonstrate that, in fact, JavaScript programs are far more amenable to analysis than previously believed. To justify our design choices, we have evaluated over 8,000 JavaScript widgets, from sources such as Live.com, Google, and the Vista Sidebar. Unlike some previous proposals [35], JavaScript_{SAFE} is entirely pragmatic, driven by what is found in real-life JavaScript widgets. Encouragingly, we have discovered that the use of `with`, `Function` and other “difficult” constructs [12] is similarly rare. In fact, `eval`, a reflective construct that usually foils static analysis, is only used in 6% of our benchmarks. However, statically unknown field references such as `a[index]`, dangerous because these can be used to get to `eval` through `this['eval']`, etc., and `innerHTML` assignments, dangerous because these can be used to inject JavaScript into the DOM, are more prevalent than previously thought. Since these features are quite common, to prevent runtime code introduction and maintain the soundness of our approach, GATEKEEPER inserts dynamic checks around statically unresolved field references and `innerHTML` assignments.

This paper contains a comprehensive large-scale experimental evaluation. To show the practicality of GATEKEEPER, we present nine representative policies for security and reliability. Our policies include restricting widgets capabilities to prevent calls to `alert` and the

use of the `XmlHttpRequest` object, looking for global namespace pollution, detecting browser redirects leading to cross-site scripting, preventing code injection, taint checking, etc. We experimented on 8,379 widgets, out of which 6,541 are analyzable by GATEKEEPER¹. Checking our nine policies resulted in us discovering a total of 1,341 verified warnings that affect 684, with only 113 false positives affecting only two widgets.

1.1 Contributions

This paper makes the following contributions:

- We propose the first points-to analysis for JavaScript programs. Our analysis is the first to handle a prototype-based language such as JavaScript. We also identify JavaScript_{SAFE}, a statically analyzable subset of the JavaScript language and propose lightweight instrumentation that restricts runtime code introduction to handle many more programs outside of the JavaScript_{SAFE} subset.
- On the basis of points-to information, we demonstrate the utility of our approach by describing nine representative security and reliability policies that are soundly checked by GATEKEEPER, meaning no false negatives are introduced. These policies are expressed in the form of succinct declarative Datalog queries. The system is highly extensible and easy to use: each policy we present is only several lines of Datalog. Policies we describe include restricting widget capabilities, making sure built-in objects are not modified, preventing code injection attempts, etc.
- Our experimental evaluation involves in excess of *eight thousand* publicly available JavaScript widgets from Live.com, the Vista Sidebar, and Google. We flag a total of 1,341 policy violations spanning 684 widgets, with 113 false positives affecting only two widgets.

¹Because we cannot ensure soundness for the remaining 1,845 widgets, we reject them without further policy checking.

1.2 Paper Organization

The rest of the paper is organized as follows. Section 2 gives an overview of our approach and summarizes the most significant analysis challenges. Section 3 provides a deep dive into the details of our analysis; a reader interested in learning about the security policies may skip this section on the first reading. Section 4 describes nine static checkers we have developed for checking security policies of JavaScript widgets. Section 5 summarizes the experimental results. Finally, Sections 6 and 7 describe related work and conclude.

2 Overview

As a recent report from Finjan Security shows, widget-based attacks are on the rise [17]. Exploits such as those in a Vista sidebar contacts widget, a Live.com RSS widget, and a Yahoo! contact widget [17, 27] not only affect unsuspecting users, they also reflect poorly on the hosting site. In a way, widgets are like operating system drivers: their quality directly affects the perceived quality of the underlying OS. While driver reliability and security has been subject of much work [7], widget security has received relatively little attention. Just like with drivers, however, widgets can run in the same page (analogous to an OS process) as the rest of the hosting site. Because widget flaws can negatively impact the rest of the site, it is our aim to develop tools to ensure widget security and reliability.

While our proposed static analysis techniques are much more general and can be used for purposes as diverse as program optimization, concrete type inference, and bug finding, the focus of this paper is on soundly enforcing security and reliability policies of JavaScript widgets. There are three principals that emerge in that scenario: the widget hosting site such as Live.com, the developer submitting a particular widget, and the user on whose computer the widget is ultimately executed. The relationship of these principals is shown in Figure 1. We are primarily interested in helping the *widget host* ensure that their users are protected.

2.1 Deployment

We envision GATEKEEPER being deployed and run by the widget hosting provider as a mandatory checking step in the online submission process, required before a widget is accepted from a widget developer. Many hosts already use captchas to ensure that the submitter is human. However, captchas say nothing about the quality and intent of the code being submitted. Using GATEKEEPER will ensure that the widget being submitted complies with the policies chosen by the host. A hosting provider has the

authority to reject some of the submitted widgets, instructing widget authors to change their code until it passes the policy checker, not unlike tools like the static driver verifier for Windows drivers [24]. Our policy checker outputs detailed information about why a particular widget fails, annotated with line numbers, which allows the widget developer to fix their code and resubmit.

2.2 Designing Static Language Restrictions

To enable sound analysis, we first restrict the input to be a subset of JavaScript as defined by the EcmaScript-262 language standard. Unlike previous proposals that significantly hamper language expressiveness for the sake of safety [13], our restrictions are relatively minor. In particular, we disallow the `eval` construct and its close cousin, the `Function` object constructor as well as functions `setTimeout` and `setInterval`. All of these constructs take a string and execute it as JavaScript code. The fundamental problem with these constructs is that they introduce new code at runtime that is unseen — and therefore cannot be reasoned about — by the static analyzer. These reflective constructs have the same expressive power: allowing one of them is enough to have the possibility of arbitrary code introduction.

We also disallow the use of `with`, a language feature that allows to dynamically substitute the symbol lookup scope, a feature that has few legitimate uses and significantly complicates static reasoning about the code. As our treatment of prototypes shows, it is in fact possible to handle `with`, but it is only used in 8% of our benchmarks. Finally, while these restrictions might seem draconian at first, they are very similar to what a recently proposed strict mode for JavaScript enforces [14].

We do allow reflective constructs `Function.call`, `Function.apply`, and the `arguments` array. Indeed, `Function.call`, the construct that allows the caller of a function to set the callee's `this` parameter, is used in 99% of Live widgets and can be analyzed statically with relative ease, so we handle this language feature. The prevalence of `Function.call` can be explained by a common coding pattern for implementing a form of inheritance, which is encouraged by Live.com widget documentation, and is found pretty much verbatim in most widgets.

In other words, our analysis choices are driven by the statistics we collect from 8,379 real-world widgets and not hypothetical considerations. More information about the relative prevalence of “dangerous” language features can be found in Figure 3. The most common “unsafe” features we have to address are `.innerHTML` assignments and statically unresolved field references. Because they are so common, we cannot simply disallow them, so we check them at runtime instead.

To implement restrictions on the allowed input, in

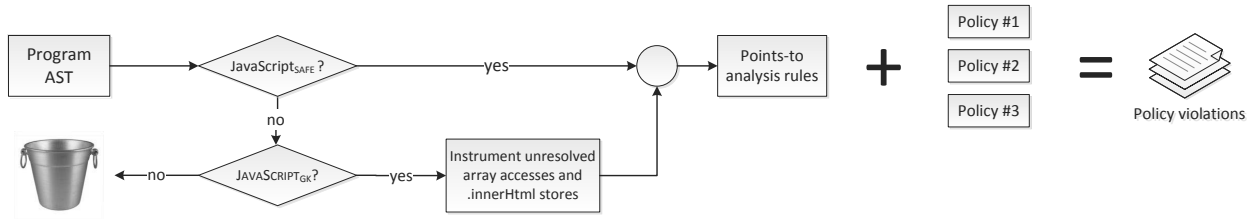


Figure 2: GATEKEEPER analysis architecture.

JavaScript Construct	Sidebar		Windows Live		Google	
	Affected	%	Affected	%	Affected	%
Non-Const Index	1,736	38.6%	176	6.5%	192	16.4%
with	422	9.4%	2	.1%	2	.2%
arguments	175	3.9%	6	.2%	3	.3%
setTimeout	824	18.3%	49	1.8%	65	5.6%
setInterval	377	8.4%	16	.6%	13	1.1%
eval	353	7.8%	10	.4%	55	4.7%
apply	173	3.8%	29	1.1%	6	.5%
call	151	3.4%	2,687	99.0%	4	.3%
Function	142	3.2%	4	.1%	21	1.8%
document.write	102	2.3%	1	0%	108	9.2%
.innerHTML	1,535	34.1%	2,053	75.6%	288	24.6%

Figure 3: Statistics for 4,501 widgets from Sidebar and 2,714 widgets from Live, and 1,171 widgets from Google.

our JavaScript parser we flag the use of lexer tokens `eval`, `Function`, and `with`, as well as `setTimeout`, and `setInterval`. We need to disallow all of these constructs because letting one of them through is enough for arbitrary code introduction. The feature we cannot handle simply using lexer token blacklisting is `document.write`. We first optimistically assume that no calls to `document.write` are present and then proceed to verify this assumption as described in Section 4.3. This way our analysis remains sound.

We consider two subsets of the JavaScript language, $\text{JavaScript}_{\text{SAFE}}$ and $\text{JavaScript}_{\text{GK}}$. The two subsets are compared in Figure 4. If the program passes the checks above *and* lacks statically unresolved array accesses and `innerHTML` assignments, it is declared to be in $\text{JavaScript}_{\text{SAFE}}$. Otherwise, these dangerous accesses are instrumented and it is declared in the $\text{JavaScript}_{\text{GK}}$ language subset. To resolve field accesses, we run a local dataflow constant propagation analysis [1] to identify the use of constants as field names. In other words, in the following code snippet

```
var fieldName = 'f';
a[fieldName] = 3;
```

the second line will be correctly converted into `a.f = 3`.

2.3 Analysis Stages

The analysis process is summarized in Figure 2. If the program is outside of $\text{JavaScript}_{\text{GK}}$, we reject it right away. Otherwise, we first traverse the program representation and output a database of facts, expressed in Datalog notation. This is basically a declarative database representing what we need to know about the input JavaScript program. We next combine these facts with a representation of the native environment of the browser discussed in Section 3.4 and the points-to analysis rules. All three are represented in Datalog and can be easily combined. We pass the result to `bddbdb`, an off-the-shelf declarative solver [33], to produce policy violations. This provides for a very agile experience, as changing the policy usually only involves editing several lines of Datalog.

2.4 Analyzing the $\text{JavaScript}_{\text{SAFE}}$ Subset

For a $\text{JavaScript}_{\text{SAFE}}$ program, we normalize each function to a set of statements shown in Figure 5. Note that the $\text{JavaScript}_{\text{SAFE}}$ language, which we shall extend in Section 3 is very much Java-like and is therefore amenable to inclusion-based points-to analysis [33]. What is not made explicit by the syntax is that $\text{JavaScript}_{\text{SAFE}}$ is a prototype-based language, not a class-based one. This means that objects do not belong to explicitly declared classes. Instead, a object creation can be based on a function, which becomes that object’s prototype. Furthermore, we support a restricted form of reflection including `Function.call`,

Feature	JavaScript _{SAFE}	JavaScript _{GK}
UNCONTROLLED CODE INJECTION		
Unrestricted eval	✗	✗
Function constructor	✗	✗
setTimeout, setInterval	✗	✗
with	✗	✗
document.write	✗	✗
Stores to code-injecting fields innerHTML, onclick, etc.	✗	✗
CONTROLLED REFLECTION		
Function.call	✓	✓
Function.apply	✓	✓
arguments array	✓	✓
INSTRUMENTATION POINTS		
Non-static field stores	✗	✓
innerHTML assignments	✗	✓

Figure 4: Support for different dynamic EcmaScript-262 language features in JavaScript_{SAFE} and JavaScript_{GK} language subsets.

$s ::=$		$\text{CALLS}(i : I, h : H)$	indicates when call site i invokes method h
ϵ	[EMPTY]	$\text{FORMAL}(h : H, z : Z, v : V)$	records formal arguments of a function
$s; s$	[SEQUENCE]	$\text{METHODRET}(h : H, v : V)$	records the return value of a method
$v_1 = v_2$	[ASSIGNMENT]	$\text{ACTUAL}(i : I, z : Z, v : V)$	records actual arguments of a function call
$v = \perp$	[PRIMASSIGNMENT]	$\text{CALLRET}(i : I, v : V)$	records the return value for a call site
return v ;	[RETURN]	$\text{ASSIGN}(v_1 : V, v_2 : V)$	records variable assignments
$v = \text{new } v_0(v_1, \dots, v_n);$	[CONSTRUCTOR]	$\text{LOAD}(v_1 : V, v_2 : V, f : F)$	represents field loads
$v = v_0(v_{\text{this}}, v_1, v_2, \dots, v_n);$	[CALL]	$\text{STORE}(v_1 : V, f : F, v_2 : V)$	represents field stores
$v_1 = v_2.f;$	[LOAD]	$\text{PTSTO}(v : V, h : H)$	represents a points-to relation for variables
$v_1.f = v_2;$	[STORE]	$\text{HEAPPTSTO}(h_1 : H, f : F, h_2 : H)$	represents a points-to relations for heap objects
$v = \text{function}(v_1, \dots, v_n) \{s; \};$	[FUNCTIONDECL]	$\text{PROTOTYPE}(h_1 : H, h_2 : H)$	records object prototypes

Figure 5: JavaScript_{SAFE} statement syntax in BNF.

Function.apply, and the arguments array. The details of pointer analysis are shown in the Datalog rules Figure 8 and discussed in detail in Section 3.

One key distinction of our approach with Java is that there is basically no distinction of heap-allocation objects and function closures in the way the analysis treats them. In other words, at a call site, if the base of a call “points to” an allocation site that corresponds to a function declaration, we statically conclude that that function might be called. While it may be possible to recover portions of the call graph through local analysis, we interleave call graph and points-to analysis in our approach.

We are primarily concerned with analyzing objects or references to them in the JavaScript heap and not primitive values such as integers and strings. We therefore do not attempt to faithfully model primitive value manipu-

Figure 6: Datalog relations used for program representation.

lation, lumping primitive values into PRIMASSIGNMENT statements.

2.5 Analysis Soundness

The core static analysis implemented by GATEKEEPER is sound, meaning that we statically provide a conservative approximation of the runtime program behavior. Achieving this for JavaScript with all its dynamic features is far from easy. As a consequence, we extend our soundness guarantees to programs utilizing a smaller subset of the language. For programs within JavaScript_{SAFE}, our analy-

$v_1 = v_2$	ASSIGN(v_1, v_2).	[ASSIGNMENT]
$v = \perp$		[BOTASSIGNMENT]
return v	CALLRET(v).	[RETURN]
<hr/>		
$v = \text{new } v_0(v_1, v_2, \dots, v_n)$	PTSTO(v, d_{fresh}). PROTOTYPE(d_{fresh}, h) : - PTSTO(v_0, m), HEAPPTSTO($m, \text{"prototype"}, h$). for $z \in \{1..n\}$, generate ACTUAL(i, z, v_z). CALLRET(i, v).	[CONSTRUCTOR]
$v = v_0(v_{\text{this}}, v_1, v_2, \dots, v_n)$	for $z \in \{1..n, \text{this}\}$, generate ACTUAL(i, z, v_z). CALLRET(i, v).	[CALL]
<hr/>		
$v_1 = v_2.f$	LOAD(v_1, v_2, f).	[LOAD]
$v_1.f = v_2$	STORE(v_1, f, v_2).	[STORE]
<hr/>		
$v = \text{function}(v_1, \dots, v_n) \{s\}$	PTSTO(v, d_{fresh}). HEAPPTSTO($d_{\text{fresh}}, \text{"prototype"}, p_{\text{fresh}}$). FUNCDECL(d_{fresh}). PROTOTYPE($p_{\text{fresh}}, h_{\text{FP}}$). for $z \in \{1..n\}$, generate FORMAL(d_{fresh}, z, v_z). METHODRET(d_{fresh}, v).	[FUNCTIONDECL]
<hr/>		

Figure 7: Datalog facts generated for each JavaScript_{SAFE} statement.

sis is sound. For programs within GATEKEEPER, our analysis is sound *as long as no code introduction is detected with the runtime instrumentation we inject*. This is very similar to saying that, for instance, a Java program is not going to access outside the boundaries of an array as long as no `ArrayOutOfBoundsException` is thrown. Details of runtime instrumentation are presented in Section 3.2. The implications of soundness is that GATEKEEPER is guaranteed to flag all policy violations, at the cost of potential false positives.

We should also point out that the GATEKEEPER analysis is inherently a *whole-program analysis*, not a modular one. The need to statically have access to the entire program is why we work so hard to limit language features that allow dynamic code loading or injection. We also generally model the runtime — or *native* — environment in which the JavaScript code executes. Our approach is sound, assuming that our native environment model is conservative. This last claim is similar to asserting that a static analysis for Java is sound, as long as *native* functions and libraries are modeled conservatively, a commonly used assumption. We also assume that the runtime instrumentation we insert is able to handle the relevant corner cases a deliberately malicious widget might try to exploit, admittedly a challenging task, as further explained in Section 3.2.

3 Analysis Details

This section is organized as follows. Section 3.1 talks about pointer analysis in detail². Section 3.2 discusses the runtime instrumentation inserted by GATEKEEPER. Section 3.3 talks about how we normalize JavaScript AST to fit into our intermediate representation. Section 3.4 talks about how we model the native JavaScript environment.

3.1 Pointer Analysis

In this paper, we describe how to implement a form of inclusion-based Andersen-style flow- and context-sensitive analysis [3] for JavaScript. It remains to be seen whether flow and context sensitivity significantly improve analysis precision; our experience with the policies in Section 4 has not shown that to be the case. We use allocation sites to approximate runtime heap objects. A key distinction of our approach in the lack of a call graph to start with: our technique allows call graph inference and points-to analysis to be interleaved. As advocated elsewhere [21], the analysis itself is expressed declaratively: we convert the program into a set of facts, to which we

²We refer the interested reader to a companion technical report [22] that discusses handling of reflective constructs `Function.call`, `Function.apply`, and `arguments`.

<i>% Basic rules</i>	
$\text{PTSTo}(v, h)$	$:- \text{ALLOC}(v, h).$
$\text{PTSTo}(v, h)$	$:- \text{FUNCDECL}(v, h).$
$\text{PTSTo}(v_1, h)$	$:- \text{PTSTo}(v_2, h), \text{ASSIGN}(v_1, v_2).$
$\text{DIRECTHEAPSTORESTo}(h_1, f, h_2)$	$:- \text{STORE}(v_1, f, v_2), \text{PTSTo}(v_1, h_1), \text{PTSTo}(v_2, h_2).$
$\text{DIRECTHEAPPOINTSTo}(h_1, f, h_2)$	$:- \text{DIRECTHEAPSTORESTo}(h_1, f, h_2).$
$\text{PTSTo}(v_2, h_2)$	$:- \text{LOAD}(v_2, v_1, f), \text{PTSTo}(v_1, h_1), \text{HEAPPTSTo}(h_1, f, h_2).$
$\text{HEAPPTSTo}(h_1, f, h_2)$	$:- \text{DIRECTHEAPPOINTSTo}(h_1, f, h_2).$
<i>% Call graph</i>	
$\text{CALLS}(i, m)$	$:- \text{ACTUAL}(i, 0, c), \text{PTSTo}(c, m).$
<i>% Interprocedural assignments</i>	
$\text{ASSIGN}(v_1, v_2)$	$:- \text{CALLS}(i, m), \text{FORMAL}(m, z, v_1), \text{ACTUAL}(i, z, v_2), z > 0.$
$\text{ASSIGN}(v_2, v_1)$	$:- \text{CALLS}(i, m), \text{METHODRET}(m, v_1), \text{CALLRET}(i, v_2).$
<i>% Prototype handling</i>	
$\text{HEAPPTSTo}(h_1, f, h_2)$	$:- \text{PROTOTYPE}(h_1, h), \text{HEAPPTSTo}(h, f, h_2).$

Figure 8: Pointer analysis inference rules for JavaScript_{SAFE} expressed in Datalog.

apply inference rules to arrive at the final call graph and points-to information.

Program representation. We define the following *domains* for the points-to analysis GATEKEEPER performs: heap-allocated objects and functions H , program variables V , call sites I , fields F , and integers Z . The analysis operates on a number of relations of fixed arity and type, as summarized in Figure 6.

Analysis stages. Starting with a set of initial input relation, the analysis follows inference rules, updating intermediate relation values until a fixed point is reached. Details of the declarative analysis and BDD-based representation can be found in [32]. The analysis proceeds in stages. In the first analysis stage, we traverse the normalized representation for JavaScript_{SAFE} shown in Figure 5. The basic facts that are produced for every statement in the JavaScript_{SAFE} program are summarized in Figure 7. As part of this traversal, we fill in relations ASSIGN , FORMAL , ACTUAL , METHODRET , CALLRET , etc. This is a relatively standard way to represent information about the program in the form of a database of facts. The second stage applies Datalog inference rules to the initial set of facts. The analysis rules are summarized in Figure 8. In the rest of this section, we discuss different aspects of the pointer analysis.

3.1.1 Call Graph Construction

As we mentioned earlier, call graph construction in JavaScript presents a number of challenges. First, unlike a language with function pointers like C, or a language with a fixed class hierarchy like Java, JavaScript does *not*

have any initial call graph to start with. Aside from local analysis, the only conservative default we have to fall back to when doing static analysis is “any call site calls every declared function,” which is too imprecise.

Instead, we chose to combine points-to and call graph constraints into a single Datalog constraint system and resolve them at once. Informally, intraprocedural data flow constraints lead to new edges in the call graph. These in turn lead to new data flow edges when we introduce constraints between newly discovered arguments and return values. In a sense, function declarations and object allocation sites are treated very much the same in our analysis. If a variable $v \in V$ may point to function declaration f , this implies that call $v()$ may invoke function f . Allocation sites and function declarations flow into the points-to relation PTSTo through relations ALLOC and FUNCDECL .

3.1.2 Prototype Treatment

The JavaScript language defines two lookup chains. The first is the lexical (or static) lookup chain common to all closure-based languages. The second is the prototype chain. To resolve o.f , we follow o ’s prototype, o ’s prototype’s prototype, etc. to locate the first object associated with field f .

Note that the object prototype (sometimes denoted as $[[\text{Prototype}]]$ in the ECMA standard) is different from the prototype field available on any object. We model $[[\text{Prototype}]]$ through the PROTOTYPE relation in our static analysis. When $\text{PROTOTYPE}(h_1, h_2)$ holds, h_1 ’s internal $[[\text{Prototype}]]$ may be h_2 ³.

³We follow the EcmaScript-262 standard; Firefox makes

Two rules in Figure 7 are particularly relevant for prototype handling: [CONSTRUCTOR] and [FUNCTIONDECL]. In the case of a constructor call, we allocate a new heap variable d_{fresh} and make the return result of the call v point to it. For (every) function m the constructor call invokes, we make sure that m 's prototype field is connected with d_{fresh} through the `PROTOTYPE` relation. We also set up `ACTUAL` and `CALLRET` values appropriately, for $z \in \{1..n\}$. In the regular [CALL] case, we also treat the `this` parameter as an extra actual parameter.

In the case of a [FUNCTIONDECL], we create two fresh allocation site, d_{fresh} for the function and p_{fresh} for the newly create prototype field for that function. We use shorthand notion h_{FP} to denote object `Function.prototype` and create a `PROTOTYPE` relation between p_{fresh} and h_{FP} . We also set up `HEAPPTSTo` relation between d_{fresh} and p_{fresh} objects. Finally, we set up relations `FORMAL` and `METHODRET`, for $z \in \{1..n\}$.

Example 1. The example in Figure 9 illustrates the intricacies of prototype manipulation. Allocation site a_1 is created on line 2. Every declaration creates a declaration object and a prototype object, such as d_T and p_T . Rules in Figure 10 are output as this code is processed, annotated with the line number they come from. To resolve the call on line 4, we need to determine what `t.bar` points to. Given `PTSTo(t, a_1)` on line 2, this resolves to the following Datalog query:

$$\text{HEAPPTSTo}(a_1, \text{"bar"}, X)?$$

Since there is nothing d_T points to *directly* by following the `bar` field, the `PROTOTYPE` relation is consulted. `PROTOTYPE(a_1, p_T)` comes from line 2. Because we have `HEAPPTSTo($p_T, \text{"bar"}, d_{bar}$)` on line 3, we resolve X to be d_{bar} . As a result, the call on line 4 may correctly invoke function `bar`. Note that our rules do not try to keep track of the order of objects in the prototype chain. \square

3.2 Programs Outside JavaScript_{SAFE}

The focus of this section is on runtime instrumentation for programs outside JavaScript_{SAFE}, but within the JavaScript_{GK} JavaScript subset that is designed to prevent runtime code introduction.

3.2.1 Rewriting .innerHTML Assignments

`innerHTML` assignments are a common dangerous language feature that may prevent GATEKEEPER from statically seeing all the code. We disallow it in JavaScript_{SAFE}, but because it is so common, we still allow it in the JavaScript_{GK} language subset. While in many cases the right-hand side of `.innerHTML` assignments is a constant,

[[Prototype]] accessible through a non-standard field `__proto__`.

there is an unfortunate coding pattern encouraged by Live widgets that makes static analysis difficult, as shown in Figure 11. The `url` value, which is the result concatenating of a constant URL and `widgetURL` is being used on the right-hand side and could be used for code injection. An assignment `v1.innerHTML = v2` is rewritten as

```
if (__IsUnsafe(v2)) {
    alert("Disguised eval attempt at <file>:<line>");
} else {
    v1.innerHTML = v2;
}
```

where `__IsUnsafe` disallows all but very simple HTML. Currently, `__IsUnsafe` is implemented as follows:

```
function __IsUnsafe(data) {
    return (toStaticHTML(data)===data);
}
```

`toStaticHTML`, a built-in function supported in newer versions of Internet Explorer, removes attempts to introduce script from a piece of HTML. An alternative is to provide a parser that allows a subset of HTML, an approach that is used in WebSandbox [25]. The call to `alert` is optional — it is only needed if we want to warn the user. Otherwise, we may just omit the statement in question.

3.2.2 Rewriting Unresolved Heap Loads and Stores

That syntax for JavaScript_{GK} supported by GATEKEEPER has an extra variant of `LOAD` and `STORE` rules for associative arrays, which introduce Datalog facts shown below:

$$\begin{array}{lll} v_1 = v_2[*] & \text{LOAD}(v_1, v_2, _) & [\text{ARRAYLOAD}] \\ v_1[*] = v_2 & \text{STORE}(v_1, _, v_2) & [\text{ARRAYSTORE}] \end{array}$$

When the indices of an associative array operation cannot be determined statically, we have to be conservative. This means that any field that may be reached can be accessed. This also means that to be conservative, we must consider the possibility that *any* field may be affected as well: the field parameter is unconstrained, as indicated by an `_` in the Datalog rules above.

Example 2. Consider the following motivating example:

```
1. var a = {
2.   'f' : function(){...},
3.   'g' : function(){...}, ...};
5. a[x + y] = function(){...};
6. a.f();
```

If we cannot statically decide which field of object `a` is being written to on line 5, we have to conservatively assume

```

1. function T(){ this.foo = function(){ return 0}};     $d_T, p_T$ 
2. var t = new T();                                   $a_1$ 
3. T.prototype.bar = function(){ return 1; };          $d_{\text{bar}}, p_{\text{bar}}$ 
4. t.bar(); // return 1

```

Figure 9: Prototype manipulation example.

1. $\text{PTSTo}(T, d_T). \text{HEAPPTSTo}(d_T, \text{"prototype"}, p_T). \text{PROTOTYPE}(p_T, h_{\text{FP}}).$
2. $\text{PTSTo}(t, a_1). \text{PROTOTYPE}(a_1, p_T).$
3. $\text{HEAPPTSTo}(p_T, \text{"bar"}, d_{\text{bar}}). \text{HEAPPTSTo}(d_{\text{bar}}, \text{"prototype"}, p_{\text{bar}}). \text{PROTOTYPE}(p_{\text{bar}}, h_{\text{FP}}).$

Figure 10: Rules created for the prototype manipulation example in Figure 9.

that the assignment could be to field f . This can affect which function is called on line 6. \square

Moreover, any statically unresolved store may introduce additional code through writing to the `innerHTML` field that will be never seen by static analysis. We rewrite statically unsafe stores $v_1[i] = v_2$ by blacklisting fields that may lead to code introduction:

```

if (i==="onclick" || i==="onkeypress" || ...) {
    alert("Disguised eval attempt at <file>:<line>");
} else
if (i==="innerHTML" && __IsUnsafe(v2)){
    alert("Unsafe innerHTML at <file>:<line>");
} else {
    v1[i] = v2;
}

```

Note that we use `===` instead of `==` because the latter form will try to coerce `i` to a string, which is not our intention. Also note that it's impossible to introduce a TOCTOU vulnerability of having `v2` change “underneath us” after the safety check because of the single-threaded nature of JavaScript.

Similarly, statically unsafe loads of the form $v_1 = v_2[i]$ can be restricted as follows:

```

if (i==="eval" || i==="setInterval" ||
    i==="setTimeout" || i==="Function" ||...)
{
    alert("Disguised eval attempt at <file>:<line>");
} else {
    v1 = v2[i];
}

```

Note that we have to check for unsafe functions such as `eval`, `setInterval`, etc. While we reject them as tokens for `JavaScriptSAFE`, they may still creep in through statically unresolved array accesses. Note that to preserve the soundness of our analysis, care must be taken to keep the blacklist comprehensive.

While we currently use a blacklist and do our best to keep it as complete as we can, ideally blacklist design and browser runtime design would go hand-in-hand. We really could benefit from a browser-specified form of runtime safety, as illustrated by the `use strict` pragma [14]. A conceptually safer, albeit more restrictive, approach is to resort to a whitelist of allowed fields.

3.3 Normalization Details

In this section we discuss several aspects of normalizing the JavaScript AST. Note that certain tricky control flow and reflective constructs like `for...in` are omitted here because our analysis is flow-insensitive.

Handling the global object. We treat the global object explicitly by introducing a variable `global` and then assigning to its fields. One interesting detail is that global variable reads and writes become *loads* and *stores* to fields of the global object, respectively.

Handling of this argument in function calls. One curious feature of JavaScript is its treatment of the `this` keyword, which is described in section 10.2 of the EcmaScript-262 standard. For calls of the form `f(x,y,...)`, the `this` value is set by the runtime to the global object. This is a pretty surprising design choice, so we translate syntactic forms `f(x,y,...)` and `o.f(x,y,...)` differently, passing the global object in place of `this` in the former case.

3.4 Native Environment

The browser embedding of the JavaScript engine has a large number of pre-defined objects. In addition to `Array`, `Date`, `String`, and other objects defined by the EcmaScript-262 standard, the browser defines objects such as `Window` and `Document`.

Native environment construction. Because we are doing whole-program analysis, we need to create stubs for

```

this.writeWidget = function(widgetURL) {
    var url = "http://widgets.clearspring.com/csproduct/web/show/flash?
    opt=-MAX/1/-PUR/http%253A%252F%252Fwww.microsoft.com&url="+widgetURL;

    var myFrame = document.createElement("div");
    myFrame.innerHTML = '<iframe id="widgetIFrame" scrolling="no"
    frameborder="0" style="width:100%;height:100%;border:0px" src="'+
    url+'"></iframe>';
    ...
}

```

Figure 11: innerHTML assignment example

the native environment so that calls to built-in methods resolve to actual functions. We recursively traverse the native embedding. For every function we encounter, we provide a default stub function(){return undefined;}. The resulting set of declarations looks as follows:

```

var global = new Object();
// this references in the global namespace refer to global
var this = global;
global.Array = new Object();
global.Array.constructor = new function(){return undefined;};
global.Array.join = new function(){return undefined;};
...

```

Note that we use an explicit global object to host a namespace for our declarations instead of the implicit this object that JavaScript uses. In most browser implementations, the global this object is aliased with the window object, leading to the following declaration: global.window = global;.

Soundness. However, as it turns out, creation of a *sound* native environment is more difficult than that. Indeed, the approach above assumes that the built-in functions return objects that are never aliased. This fallacy is most obviously demonstrated by the following code:

```

var parent_div = document.getElementById('header');
var child_div = document.createElement('div');
parent_div.appendChild(child_div);
var child_div2 = parent_div.childNodes[0];

```

In this case, child_div and child_div2 are aliases for the same DIV element. if we pretend they are not, we will miss an existing alias. We therefore model operations such as appendChild, etc. in JavaScript code, effectively creating *mock-ups* instead of native browser-provided implementations.

In our implementation, we have done our best to ensure the soundness of the environment we produce by starting with an automatically generated collection of stubs and augmenting them by hand to match what we believe the proper browser semantics to be. This is similar to modeling memcpy in a static analysis of C code or native methods in a static analysis for Java. However, as with two instance of foreign function interface (FFI) modeling above, this form of manual involvement is often error-

prone. It many also unfortunately compromise the soundness of the overall approach, both because of implementation mistakes and because of browser incompatibilities. A potential alternative to our current approach and part of our future work is to consider a standards-compliant browser that that implements some of its library code in JavaScript, such as Chrome. With such an approach, because libraries become amenable to analysis, the need for manually constructed stubs would be diminished.

When modeling the native environment, when in doubt, we tried to err on the side of caution. For instance, we do not attempt to model the DOM very precisely, assuming initially that any DOM-manipulating method may return any DOM node (effectively all DOM nodes are statically modeled as a single allocation site). Since our policies in Section 4 do not focus on the DOM, this imprecise, but sound modeling does not result in false positives.

4 Security and Reliability Policies

This section is organized as follows. Sections 4.1–4.4 talk about six policies that apply to widgets from all widgets hosts we use in this paper (Live, Sidebar, and Google). Section 4.5 talks about host-specific policies, where we present two policies specific to Live and one specific to Sidebar widgets. Along with each policy, we present the Datalog query that is designed to find policy violations. We have run these queries on our set of 8,379 benchmark widgets. A detailed discussion of our experimental findings can be found in Section 5.

4.1 Restricting Widget Capabilities

Perhaps the most common requirement for a system that reasons about widgets is the ability to restrict code capabilities, such as disallowing calling a particular function, using a particular object or namespace, etc. The Live Widget Developer Checklist provides many such examples [34]. This is also what systems like Caja and Web-Sandbox aim to accomplish [25, 29]. We can achieve the same goal statically.

Pop-up boxes represent a major annoyance when using

```
Array.prototype.feed = function(o, s){
    if(!s){s=o;o={};}
    var k,p=s.split(":");
    while(typeof(k=p.shift())!="undefined")
        o[k]=this.shift();
    return o;
}
```

$$\begin{aligned} GlobalSym(m, h) &:- \text{PTSTO}(\text{"global"}, g), \\ &\quad \text{HEAPPTSTO}(g, m, h). \\ AlertCalls(i) &:- GlobalSym(\text{"alert"}, h), \\ &\quad \text{CALLS}(i, h). \end{aligned}$$

4.3 Detecting Code Injection

```
var x = document;
var y = x.write;
y("<script>alert('hi');</script>");
```

The query below showcases the power of points-to analysis. In addition to finding the direct calls, the query below will correctly determine that the call to `y` invokes `document.write`.

Query output: $DocumentWrite(i : I)$

$$\begin{aligned} \text{DocumentWrite}(i) &:- \text{GlobalSym}(\text{"document"}, d), \\ &\quad \text{HEAPPTSTo}(d, \text{"write"}, m), \\ &\quad \text{CALLS}(i, h). \\ \text{DocumentWrite}(i) &:- \text{GlobalSym}(\text{"document"}, d), \\ &\quad \text{HEAPPTSTo}(d, \text{"writeln"}, m), \\ &\quad \text{CALLS}(i, h). \end{aligned}$$

4.4 Redirecting the Browser

JavaScript in the browser has write access to the current page's location, which may be used to redirect the user to a malicious site. Google widget `Google.Calculator` performing such redirection is shown below:

```
window.location =  
    "http://e-r.se/google-calculator/index.htm"
```

Allowing such redirect not only opens the door to phishing widgets luring users to malicious sites, redirects within an iframe also open the possibility of running code that has not been adequately checked by the hosting site, potentially circumventing policy checking entirely. Another concern is cross-site scripting attacks that involve stealing cookies: a cross-site scripting attack may be mounted by assigning a location of the form "http://www.evil.com/" + document.cookie. Of

Query output: *FrozenViolation(v : V)*

```
BuiltInObject(h) :- GlobalSym("Boolean", h).   BuiltInObject(h) :- GlobalSym("Array", h).
BuiltInObject(h) :- GlobalSym("Date", h).       BuiltInObject(h) :- GlobalSym("Function", h).
BuiltInObject(h) :- GlobalSym("Math", h).       BuiltInObject(h) :- GlobalSym("Document", h).
BuiltInObject(h) :- GlobalSym("Window", h).

FrozenViolation(v) :- STORE(v, _, _), PTSTo(v, h), BuiltInObject(h).
```

Figure 12: FrozenViolations query

course, `grep` is not an adequate tool for spotting redirects, both because of the aliasing issue described above and because *read access* to `window.location` is in fact allowed. Moreover, redirects can take many forms, which we capture through the queries below. Direct location assignment are found by the following query:

Query output: *LocationAssign(v : V)*

```
LocationAssign(v) :- GlobalSym("window", h),
                     PTSTo(v, h),
                     STORE(_, "location", v).

LocationAssign(v) :- GlobalSym("document", h),
                     PTSTo(v, h),
                     STORE(_, "location", v).

LocationAssign(v) :- PTSTo("global", h),
                     PTSTo(v, h),
                     STORE(_, "location", v).
```

Storing to location object's properties are found by the following query:

```
LocationAssign(v) :- GlobalSym(h, "location"),
                     PTSTo(v, h),
                     STORE(v, _, _).
```

Calling methods on the location object are found by the following query:

Query output: *LocationChange(i : I)*

```
LocationChange(i) :- LocationObject(h),
                     HEAPPTSTo(h, "assign", h'),
                     CALLS(i, h').

LocationChange(i) :- LocationObject(h),
                     HEAPPTSTo(h, "reload", h'),
                     CALLS(i, h').

LocationChange(i) :- LocationObject(h),
                     HEAPPTSTo(h, "replace", h'),
                     CALLS(i, h').
```

```
var SearchTag = new String ("Home");
var SearchTagStr = new String(
    "meta%3ASearch.tag%28%22beginTag+" +
    SearchTag + "endTag%22%29");
var QnaURL = new String(
    SearchHostPath /** SearchQstateStr */+
    SearchTagStr + "&format=rss" );

// define the constructor for your Gadget
Microsoft.LiveQnA.RssGadget =
    function(p_elSource, p_args, p_namespace) { ... }
```

Figure 13: Example of a global namespace pollution violation (Section 4.5.2) in a Live widget.

Function `window.open` is another form of redirects, as the following query shows:

Query output: *WindowOpen(i : I)*

```
WindowOpen(i) :- WindowObject(h),
                 HEAPPTSTo(h, "open", h'),
                 CALLS(i, h').
```

4.5 Host-specific Policies

The policies we have discussed thus far have been relatively generic. In this section, we give examples of policies that are specific to the host site they reside on.

4.5.1 No XMLHttpRequest Use in Live Widgets

The first policy of this sort comes directly from the Live Web Widget Developer Checklist [34]. Among other rules, they disallow the use of `XMLHttpRequest` object in favor of function `Web.Network.createRequest`. The latter makes sure that the network requests are properly proxied so they can work cross-domain:

Query output: *XMLHttpRequest(i : I)*

```
XMLHttpRequest(i) :- GlobalSym("XMLHttpRequest", h),
                     CALLS(i, h).
```

Query output: *ActiveXExecute*(*i* : *I*)

ActiveXObjectCalls(*i*) :- *GlobalSym*("ActiveXObject", *h'*), *CALLS*(*i*, *h'*).

ShellExecuteCalls(*i*) :- *PTSTo*("global", *h*₁), *HEAPPTSTo*(*h*₁, "System", *h*₂),
 HEAPPTSTo(*h*₂, "Shell", *h*₃), *HEAPPTSTo*(*h*₃, "execute", *h*₄), *CALLS*(*i*, *h*₄).

ActiveXExecute(*i*) :- *ActiveXObjectCalls*(*i*), *CALLRET*(*i*, *v*), *PTSTo*(*v*, *h*),
 HEAPPTSTo(*h*, $_$, *m*), *CALLS*(*i*^{*}, *m*), *CALLRET*(*i*^{*}, *r*), *PTSTo*(*r*, *h*^{*}),
 ShellExecuteCalls(*i*[']), *ACTUAL*(*i*['], $_$, *v*[']), *PTSTo*(*v*['], *h*^{*}).

Figure 14: Query for finding information flow violations in Vista Sidebar widgets.

4.5.2 Global Namespace Pollution in Live Widgets

Because web widgets can be deployed on a page with other widgets running within the same JavaScript interpreter, polluting the global namespace, leading to name clashes and unpredictable behavior. This is why hosting providers such as Facebook, Yahoo!, Live, etc. strongly discourage pollution of the global namespace, favoring a module or a namespace approach instead [11] that avoids name collision. We can easily prevent stores to the global scope:

Query output: *GlobalStore*(*h* : *H*)

GlobalStore(*h*) :- *PTSTo*("global", *g*),
 HEAPPTSTo(*g*, $_$, *h*).

An example of a violation of this policy from a Live.com widget is shown in Figure 13. Because the same widget can be deployed twice within the same interpreter scope with different values of *SearchTag*, this can lead to a data race on the globally declared variable *SearchTagStr*.

Note that our analysis approach is radically different from proposals that advocate language restrictions such as AdSafe or Cajita [12, 13, 29] to protect access to the global object. The difficulty those techniques have to overcome is that the *this* identifier in the global scope will point to the global object. However, disallowing *this* completely makes object-oriented programming difficult. With the whole-program analysis GATEKEEPER implements, we do not have this problem. We are able to distinguish references to *this* that point to the global object (aliased with the *global* variable) from a local reference to *this* within a function.

4.5.3 Tainting Data in Sidebar Widgets

This policy ensures that data from ActiveX controls that may be instantiated by a Sidebar widget does not get passed into *System.Shell.execute* for direct execution on the user's machine. This is because it is common for ActiveX controls to retrieve unsanitized network data, which

is how a published RSS Sidebar exploit operates [27]. There, data obtained from an ActiveX-based RSS control was assigned directly to the *innerHTML* field withing a widget, allowing a cross-site scripting exploit. What we are looking for is demonstrated by the pattern:

```
var o = new ActiveXObject();
var x = o.m();
System.Shell.Execute(x);
```

The Datalog query in Figure 14 looks for instances where the tainted result of a call to method *m* on an ActiveX object is directly passed as an argument to the “sink” function *System.Shell.Execute*.

Auxiliary queries *ActiveXObjectCalls* and *ShellExecuteCalls* look for source and sink calls and *ShellExecuteCalls* ties all the constraints together, effectively matching the call pattern described above. As previously shown for the case of Java information flow [23], similar queries may be used to find information flow violations that involve cookie stealing and location resetting, as described in Chugh et al. [10].

5 Experimental Results

For our experiments, we have downloaded a large number of widgets from widget hosting sites' widget galleries. As mentioned before, we have experimented with widgets from Live.com, the Vista Sidebar, and Google. We automated the download process to save widgets locally for analysis. Once downloaded, we parsed through each widget's manifesto to determine where the relevant JavaScript code resides. This process was slightly different across the widget hosts. In particular, Google widgets tended to embed their JavaScript in HTML, which required us to develop a limited-purpose HTML parser. In the Sidebar case, we had to extract the relevant JavaScript code out of an archive. At the end of this process, we ended up with a total of 8,379 JavaScript files to analyze.

Figure 15 provides aggregate statistics for the widgets we used as benchmarks. For each widget source,

Widget Source	Avg. LOC	Count	Widget counts			
			JavaScript _{GK}		JavaScript _{SAFE}	
Live.com	105	2,707	2,643	97%	643	23%
Vista sidebar	261	4,501	2,946	65%	1,767	39%
Google.com/ig	137	1,171	962	82%	768	65%

Figure 15: Aggregate statistics for widgets from Live portal, Windows Sidebar, and Google portal widget repositories (columns 2–3). Information about widget distribution for different JavaScript language subsets (columns 4–7).

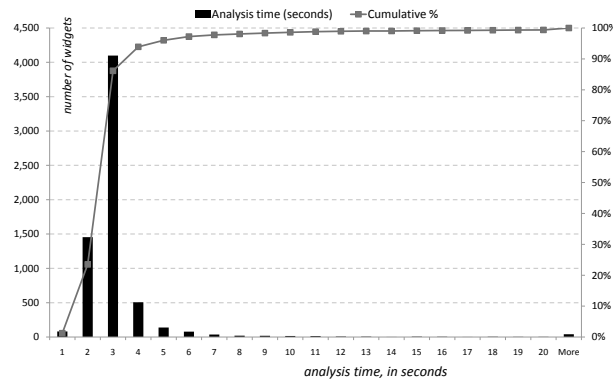


Figure 17: Histogram showing GATEKEEPER processing times.

we specify the total number of widgets we managed to obtain in column 2. Column 3 shows the average lines-of-code count for every widget. In general, Sidebar widgets tend to be longer and more involved than their Web counterparts, as reflected in the average line of code metric. Note that in addition to every widget’s code, at the time of policy checking, we also prepend the native environment constructed as described in Section 3.4. The native environment constitutes 270 lines of non-comment JavaScript code (127 for specifying the browser embedding and 143 for specifying built-in objects such as `Array` and `Date`).

5.1 Result Summary

A summary of our experimental results is presented in Figure 16. For each policy described in Section 4, we show the the total number of violations across 8,379 benchmarks, and the number of violating benchmarks. The latter two may be different because there could be several violations of a particular query per widget. We also show the percentage of benchmarks for which we find policy violations. As can be seen from the table, overall, policy violations are quite uncommon, with only several percent of widgets affected in each case. Overall, a total of 1,341 policy violations are reported.

As explained in Section 4.5, we only ran those policies on the appropriate subset of widgets, leaving other table

```
function MM_preloadImages() {
  var d=m_Doc;
  if(d.images){
    if(!d.MM_p) d.MM_p=new Array();
    var i,j=d.MM_p.length,
    a=MM_preloadImages.arguments;
    for(i=0; i<a.length; i++){
      if (a[i].indexOf("#")!=0){
        d.MM_p[j]=new Image;
        d.MM_p[j++].src=a[i];
      }
    }
  }
}
```

Figure 18: False positives in `common.js` from JustMusic.FM.

cells blank. To validate the precision of our analysis, we have examined all violations reported by our policies. For examination, GATEKEEPER output was cross-referenced with widget sources. Luckily for us, most of our query results were easy to spot-check by looking at one or two lines of corresponding source code, which made result checking a relatively quick task. Encouragingly, for most inputs, GATEKEEPER was quite precise.

5.2 False Positives

We should point out that a conservative analysis such as GATEKEEPER is inherently imprecise. Two main sources of false positives in our formulation are prototype handling and arrays. Only two widgets out of over 6,000 analyzed files in the JavaScript_{GK} subset lead to false positives in our experiments. Almost all false positive reports come from the Sidebar widget, JustMusic.FM, file `common.js`. Because of our handling of arrays, the analysis conservatively concludes that certain heap-allocated objects can reach many others by following *any* element of array `a`, as shown in Figure 18. In fact, this example is contains a number of features that are difficult to analyze statically: array aliasing, the use of `arguments` array, as well as array element loads and stores, so it is not entirely surprising that their combination leads to imprecision.

It is common for a single imprecision within static analysis to create numerous “cascading” false positive reports. This is the case here as well. Luckily, it is possible to group cascading reports together in order to avoid overwhelming the user with false positives caused by a single imprecision. This imprecision in turn affects *FrozenViolation* and *LocationAssign* queries leading to many very similar reports. A total of 113 false positives are reported, but luckily they affect only two widgets.

5.3 Analysis Running Times

Our implementation uses a publicly available declarative analysis engine provided by bddb [32]. This is a

		LIVE WIDGETS				VISTA SIDEBAR				GOOGLE WIDGETS						
Query	Section	Viol.	Affected	%	FP	Affected	Viol.	Affected	%	FP	Affected	Viol.	Affected	%	FP	Affected
AlertCalls(i : I)	4.1	54	29	1.1	0	0	161	84	2.9	0	0	57	35	3.6	0	0
FrozenViolation(v : V)	4.2	3	3	0.1	0	0	143	52	1.5	94	1	1	1	0.1	0	0
DocumentWrite(i : I)	4.3	5	1	0.0	0	0	175	75	1.7	0	0	158	88	8.1	0	0
LocationAssign(v : V)	4.4	3	3	0.1	2	1	157	109	3.8	15	1	9	9	0.7	0	0
LocationChange(i : I)	4.4	3	3	0.1	0	0	21	20	0.7	1	1	3	3	0.3	0	0
WindowOpen(i : I)	4.4	50	22	0.9	0	0	182	87	3.0	1	1	19	14	1.5	0	0
XMLHttpRequest(i : I)	4.5	1	1	0.0	0	0	—	—	—	—	—	—	—	—	—	—
GlobalStore(v : V)	4.5	136	45	1.7	0	0	—	—	—	—	—	—	—	—	—	—
ActiveXExecute(i : I)	4.5	—	—	—	—	—	0	0	0	0	0	—	—	—	—	—

Figure 16: Experimental result summary for nine policies described in Section 4. Because some policies are host-specific, we only run them on a subset of widgets. “—” indicates experiments that are not applicable.

	Live	Sidebar	Google
Number of instrumented files	2,000	1,179	194
Instrumentation points per file	1.74	8.86	5.63
Estimated overhead	40%	65%	73%

Figure 19: Instrumentation statistics.

highly optimized BDD-based solver for Datalog queries used for static analysis in the past. Because repeatedly starting `bddbddb` is inefficient we perform both the points-to analysis *and* run our Datalog queries corresponding to the policies in Section 4 as part of one run for each widget.

Our analysis is quite scalable in practice, as shown in Figure 17. This histogram shows the distribution of analysis time, in seconds. These results were obtained on a Pentium Core 2 duo 3 GHz machine with 4 GB of memory, running Microsoft Vista SP1. Note that the analysis time includes the JavaScript parsing time, the normalization time, the points-to analysis time, and the time to run all nine policies. For the vast majority of widgets, the analysis time is under 4 seconds, as shown by the cumulative percentage curve in the figure. The `bddbddb`-based approach has been shown to scale to much larger programs — up to 500,000 lines of code — in the past [32], so we are confident that we should be able to scale to larger codebases in GATEKEEPER as well.

5.4 Runtime Instrumentation

Programs outside of the JavaScript_{SAFE} language subset but within the JavaScript_{CK} language subset require instrumentation. Figure 19 summarizes data on the number of instrumentation points required, both as an absolute number and in proportion of the number of widgets that required instrumentation.

We plan to fully assess our runtime overhead as part of future work. However, we do not anticipate it to be pro-

hibitively high. The number of instrumentation points per instrumented widget ranges roughly in proportion to the size and complexity of the widget. However, it is generally difficult to perform large-scale overhead measurements for a number of highly interactive widgets.

Instead we have devised an experiment to approximate the overheads. Note that we can discern the average density of checks from the numbers in Figure 19: for instance, for Live.com, the number of instrumentation points per file is 1.74, with an average file being 105 lines, as shown in Figure 15. This yields about 2% of all lines being instrumented, on average.

To mimic this runtime check density, we generate a test script shown in Figure 20 with 100 fields stores, where the first two stores require runtime checking and the other 98 are statically known. For Sidebar and Google widgets, we construct similar test scripts with a different density of checks. As shown below, we use index `innerHTML` for one out of two rewritten cases for Live. We use it for 2 out of 3 cases for Sidebase, and 2 out of 4 cases for Google. This represents a pretty high frequency of `innerHTML` assignments.

We wrap this code in a loop that we run 1,000 times to be able to measure the overheads reliably and then take the median over several runs to account for noise. The baseline is the same test with no index or right-hand side checks. We observe overheads ranging between 40–73% across the different instrumentation densities, as shown in Figure 19. It appears that calls to `toStaticHTML` result in a pretty substantial runtime penalty. This is likely because the relatively heavy-weight HTML parser of the browser needs to be invoked on every HTML snippet.

Note that this experiment provides an approximate measure of overhead that real programs are likely to experience. However, these numbers are encouraging, as they are significantly smaller overheads on the order of 6–40x that tools like Caja may induce [28].

```

console.log(new Date().getTime());
var v1 = new Array();
var v2 = "<div onclick='alert(38);'>" +
  "<h2>Hello<script>alert(38)</script></div>";
for(var iter = 0; iter < 1000; iter++){
  // first store: check
  var i = 'innerHTML';
  if (i=="onclick" || i=="onkeypress" || ...) {
    alert("Disguised eval at <file>:<line>");
  } else
  if(i=="innerHTML" && __IsUnsafe(v2)){
    alert("Unsafe innerHTML at <file>:<line>");
  } else {
    v1[i] = v2;
  }

  // second store: check
  i = 'onclick';
  if (i=="onclick" || i=="onkeypress" || ...) {
    alert("Disguised eval at <file>:<line>");
  } else
  if(i=="innerHTML" && __IsUnsafe(v2)){
    alert("Unsafe innerHTML at <file>:<line>");
  } else {
    v1[i] = v2;
  }

  // all other stores are unchecked
  v1[i] = 2;
  v1[i] = 3;
  ...
  v1[i] = 100;
}
console.log(new Date().getTime());

```

Figure 20: Measuring the overhead of GATEKEEPER checking.

6 Related Work

Much of the work related to this paper focuses on limiting various attack vectors that exist in JavaScript. They do this through the use of type systems, language restrictions, and modifications to the browser or the runtime. We describe these strategies in turn below.

6.1 Static Safety Checks

JavaScript is a highly dynamic language which makes it difficult to reason about programs written in it. However, with certain expressiveness restrictions, desirable security properties can be achieved. ADSafe and Facebook both implement a form of static checking to ensure a form of safety in JavaScript code. ADSafe [13] disallows dynamic content, such as `eval`, and performs static checking to ensure the JavaScript in question is safe. Facebook takes an approach similar to ours in rewriting statically unresolved field stores, however, it appears that, unlike GATEKEEPER, they do not try to do local static analysis of field names. Facebook uses a JavaScript language variant called FBJS [15], that is like JavaScript in many ways,

but DOM access is restricted and all variable names are prefixed with a unique identifier to prevent name clashes with other FBJS programs on the same page.

In many ways, however, designing a safe language subset is a tricky business. Until recently, it was difficult to write anything but most simple applications in AdSafe because of its static restrictions, at least in our personal experience. More recently, AdSafe was updated with APIs to lift some of initial restrictions and allow DOM access, etc., as well as several illustrative sample widgets. Overall, these changes to allow compelling widgets to be written are an encouraging sign. While quite expressive, FBJS has been the subject of several well-publicised attacks that circumvent the isolation of the global object offered through Facebook sandbox rewriting [2]. This demonstrates that while easy to implement, reasoning about what static language restrictions accomplish is tricky.

GATEKEEPER largely sidesteps the problem of proper language subset design, opting for whole program analysis instead. We do not try to prove that JavaScript_{SAFE} programs cannot pollute the global namespace for *all* programs, for example. Instead, we take the entire program and a representation of its environment and use our static analysis machinery to check if this may happen for the input program in question. The use of static and points-to analysis for finding and vulnerabilities and ensuring security properties has been previously explored for other languages such as C [6] and Java [23].

An interesting recent development in JavaScript language standards committees is the strict mode (use strict) for JavaScript [14], page 223, which is being proposed around the time of this writing. Strict mode accomplishes many of the goals that JavaScript_{SAFE} is designed to accomplish: `eval` is largely prohibited, bad coding practices such as assigning to the arguments array are prevented, `with` is no longer allowed, etc. Since the strict mode supports customization capabilities, going forward we hope to be able to express JavaScript_{SAFE} and JavaScript_{GK} restrictions in a standards-compliant way, so that future off-the-shelf JavaScript interpreters would be able to enforce them.

6.2 Rewriting and Instrumentation

A practical alternative to static language restrictions is instrumentation. Caja [29] is one such attempt at limiting capabilities of JavaScript programs and enforcing this through the use of runtime checks. WebSandbox is another project with similar goals that also attempts to enforce reliability and resource restrictions in addition to security properties [25].

Yu et al. traverse the JavaScript document and rewrite based on a security policy [35]. Unlike Caja and WebSandbox, they prove the correctness of their rewriting

with operational semantics for a subset of JavaScript called CoreScript. BrowserShield [30] similarly uses dynamic and recursive rewriting to ensure that JavaScript and HTML are safe, for a chosen version of safety, and all content generated by the JavaScript and HTML is also safe. Instrumentation can be used for more than just enforcing security policies. AjaxScope [20] rewrites JavaScript to insert instrumentation that sends runtime information, such as error reporting and memory leak detection, back to the content provider.

Compared to these techniques, GATEKEEPER has two main advantages. First, as a mostly static analysis, GATEKEEPER places little runtime overhead burden on the user. While we are not aware of a comprehensive overhead evaluation that has been published, it appears that the runtime overhead of Caja and WebSandbox may be high, depending on the level of rewriting. For instance, a Caja authors' report suggest that the overhead of various subsets that are part of Caja are 6–40x [28]. Second, as evidenced by the Facebook exploits mentioned above [2], it is challenging to reason about whether source-level rewriting provides complete isolation. We feel that sound static analysis may provide a more systematic way to reason about what code can do, especially in the long run, as it pertains to issues of security, reliability, and performance. While the soundness of the native environment and exhaustiveness of our runtime checks might be weak points of our approach, we feel that we can address these challenges as part of future work.

6.3 Runtime and Browser Support

Current browser infrastructure and the HTML standard require a page to fully trust foreign JavaScript if they want the foreign JavaScript to interact with their site. The alternative is to place foreign JavaScript in an isolated environment, which disallows any interaction with the hosting page. This leads to web sites trusting untrustworthy JavaScript code in order to provide a richer web site. One solution to get around this all-or-nothing trust problem is to modify browsers and the HTML standard to include a richer security model that allows untrusted JavaScript controlled access to the hosting page.

MashupOS [18] proposes a new browser that is modeled after an OS and modifies the HTML standard to provide new tags that make use of new browser functionality. They provide rich isolation between execution environments, including resource sharing and communication across instances. In a more lightweight modification to the browser and HTML, Felt et al. [16] add a new HTML tag that labels a `div` element as untrusted and limits the actions that any JavaScript inside of it can take. This would allow content providers to create a sand box in which to place untrusted JavaScript. Integrating GATE-

KEEPER techniques into the browser itself, without relying on server-side analysis, and making them fast enough for daily use, is part of future work.

6.4 Typing and Analysis of JavaScript

A more useful type system in JavaScript could prevent errors or safety violations. Since JavaScript does not have a rich type system to begin with, the work here is devising a correct type system for JavaScript and then building on the proposed type system. Soft typing [8] might be one of the more logical first steps in a type system for JavaScript. Much like dynamic rewriters insert code that must be executed to ensure safety, soft typing must insert runtime checks to ensure type safety.

Other work has been done to devise a static type system that describes the JavaScript language [4, 5, 31]. These works focus on a subset of JavaScript and provide sound type systems and semantics for their restricted subsets of JavaScript. As far as we can tell, none of these approaches have been applied to realistic bodies of code. GATEKEEPER uses a pointer analysis to reason about the JavaScript program in contrast to the type systems and analyses of these works. We feel that the ability to reason about pointers and the program call graph allows us to express more interesting security policies than we would be able otherwise.

A contemporaneous project by Chugh et al. focuses on staged analysis of JavaScript and finding information flow violations in client-side code [10]. Chugh et al. focus on information flow properties such as reading document cookies and changing the locations, not unlike the location policy described in Section 4.4. A valuable feature of that work is its support for dynamically loaded and generated JavaScript in the context of what is generally thought of as whole-program analysis.

7 Conclusions

This paper presents GATEKEEPER, a mostly static sound policy enforcement tool for JavaScript programs. GATEKEEPER is built on top of what to our knowledge is the first pointer analysis developed for JavaScript. To show the practicality of our approach, we describe nine representative security and reliability policies for JavaScript widgets. Statically checking these policies results in 1,341 verified warnings in 684 widgets, with 113 false positives affecting only two widgets.

We feel that static analysis of JavaScript is a key building block for enabling an environment in which code from different parties can safely co-exist and interact. The ability to analyze a programming language using automatic tools is a valuable one for long-term language success.

It is therefore our hope that our experience with analyzable JavaScript language subsets will inform the design of language restrictions build into future versions of the JavaScript language, as illustrated by the JavaScript use strict mode.

While in this paper our focus is on policy enforcement, the techniques outlined here are generally useful for any task that involves reasoning about code such as code optimization, rewriting, program understanding tools, bug finding tools, etc. Moreover, we hope that GATEKEEPER paves the way for centrally-hosted software repositories such as the iPhone application store, Windows Marketplace, or Android Market to ensure the security and quality of software contributed by third parties.

Acknowledgments

We are grateful to Trishul Chilimbi, David Evans, Karthik Pattabiraman, Nikhil Swamy, and the anonymous reviewers for their feedback on this paper. We appreciate John Whaley's help with bddbldb.

References

- [1] A. V. Aho, M. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2007.
- [2] Ajaxian. Facebook JavaScript and security. <http://ajaxian.com/archives/facebook-javascript-and-security>, Aug. 2007.
- [3] L. O. Andersen. Program analysis and specialization for the C programming language. Technical report, University of Copenhagen, 1994.
- [4] C. Anderson and P. Giannini. Type checking for JavaScript. In *In WOOD 04, volume WOOD of ENTCS*. Elsevier, 2004. <http://www.binarylord.com/work/js0wood.pdf>, 2004.
- [5] C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for JavaScript. In *In Proceedings of the European Conference on Object-Oriented Programming*, pages 429–452, July 2005.
- [6] D. Avots, M. Dalton, B. Livshits, and M. S. Lam. Improving software security with a C pointer analysis. In *Proceedings of the International Conference on Software Engineering*, pages 332–341, May 2005.
- [7] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *European Conference on Computer Systems*, pages 73–85, 2006.
- [8] R. Cartwright and M. Fagan. Soft typing. *ACM SIGPLAN Notices*, 39(4):412–428, 2004.
- [9] B. Chess, Y. T. O'Neil, and J. West. JavaScript hijacking. www.fortifysoftware.com/servlet/downloads/public/Javascript.Hijacking.pdf, Mar. 2007.
- [10] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for JavaScript. In *Proceedings of the Conference on Programming Language Design and Implementation*, June 2009.
- [11] D. Crockford. Globals are evil. <http://yuiblog.com/blog/2006/06/01/global-domination/>, June 2006.
- [12] D. Crockford. *JavaScript: the good parts*. 2008.
- [13] D. Crockford. AdSafe: Making JavaScript safe for advertising. <http://www.adsafe.org>, 2009.
- [14] ECMA. Ecma-262: Ecma/tc39/2009/025, 5th edition, final draft. <http://www.ecma-international.org/publications/files/drafts/tc39-2009-025.pdf>, Apr. 2009.
- [15] Facebook, Inc. Fbjs. <http://wiki.developers.facebook.com/index.php/FBJS>, 2007.
- [16] A. Felt, P. Hooimeijer, D. Evans, and W. Weimer. Talking to strangers without taking their candy: isolating proxied content. In *Proceedings of the Workshop on Social Network Systems*, pages 25–30, 2008.
- [17] Finjan Inc. Web security trends report. <http://www.finjan.com/GetObject.aspx?ObjId=506>.
- [18] J. Howell, C. Jackson, H. J. Wang, and X. Fan. MashupOS: Operating system abstractions for client mashups. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, May 2007.
- [19] javascript-reference.info. JavaScript obfuscators review. <http://javascript-reference.info/javascript-obfuscators-review.htm>, 2008.
- [20] E. Kiciman and B. Livshits. AjaxScope: a platform for remotely monitoring the client-side behavior of Web 2.0 applications. In *Proceedings of Symposium on Operating Systems Principles*, Oct. 2007.
- [21] M. S. Lam, J. Whaley, B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel. Context-sensitive program analysis as database queries. In *Proceedings of the Symposium on Principles of Database Systems*, June 2005.
- [22] B. Livshits and S. Guarnieri. Gatekeeper: Mostly static enforcement of security and reliability policies for JavaScript code. Technical Report MSR-TR-2009-43, Microsoft Research, Feb. 2009.
- [23] B. Livshits and M. S. Lam. Finding security errors in Java programs with static analysis. In *Proceedings of the Usenix Security Symposium*, pages 271–286, Aug. 2005.
- [24] Microsoft Corporation. Static driver verifier. <http://www.microsoft.com/whdc/devtools/tools/SDV.mspx>, 2005.
- [25] Microsoft Live Labs. Live Labs Websandbox. <http://websandbox.org>, 2008.
- [26] Microsoft Live Labs. Quality of service (QoS) protections. <http://websandbox.livelabs.com/documentation/use-qos.aspx>, 2008.
- [27] Microsoft Security Bulletin. Vulnerabilities in Windows gadgets could allow remote code execution (938123). <http://www.microsoft.com/technet/security/Bulletin/MS07-048.mspx>, 2007.
- [28] M. S. Miller. Is it possible to mix ExtJS and google-caja to enhance security. <http://extjs.com/forum/showthread.php?p=268731#post268731>, Jan. 2009.
- [29] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized JavaScript. <http://google-caja.googlecode.com/files/caja-2007.pdf>, 2007.
- [30] C. Reis, J. Dunagan, H. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, 2006.
- [31] P. Thiemann. Towards a type system for analyzing JavaScript programs. 2005.
- [32] J. Whaley, D. Avots, M. Carbin, and M. S. Lam. Using Datalog and binary decision diagrams for program analysis. In *Proceedings of the Asian Symposium on Programming Languages and Systems*, Nov. 2005.
- [33] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 131–144, June 2004.
- [34] Windows Live. Windows live gadget developer checklist. <http://dev.live.com/gadgets/sdk/docs/checklist.htm>, 2008.
- [35] D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript instrumentation for browser security. In *Proceedings of Conference on Principles of Programming Languages*, Jan. 2007.

NOZZLE: A Defense Against Heap-spraying Code Injection Attacks

Paruj Ratanaworabhan
Cornell University
paruj@csl.cornell.edu

Benjamin Livshits
Microsoft Research
livshits@microsoft.com

Benjamin Zorn
Microsoft Research
zorn@microsoft.com

Abstract

Heap spraying is a security attack that increases the exploitability of memory corruption errors in type-unsafe applications. In a heap-spraying attack, an attacker coerces an application to allocate many objects containing malicious code in the heap, increasing the success rate of an exploit that jumps to a location within the heap. Because heap layout randomization necessitates new forms of attack, spraying has been used in many recent security exploits. Spraying is especially effective in web browsers, where the attacker can easily allocate the malicious objects using JavaScript embedded in a web page. In this paper, we describe NOZZLE, a runtime heap-spraying detector. NOZZLE examines individual objects in the heap, interpreting them as code and performing a static analysis on that code to detect malicious intent. To reduce false positives, we aggregate measurements across all heap objects and define a global heap health metric.

We measure the effectiveness of NOZZLE by demonstrating that it successfully detects 12 published and 2,000 synthetically generated heap-spraying exploits. We also show that even with a detection threshold set six times lower than is required to detect published malicious attacks, NOZZLE reports no false positives when run over 150 popular Internet sites. Using sampling and concurrent scanning to reduce overhead, we show that the performance overhead of NOZZLE is less than 7% on average. While NOZZLE currently targets heap-based spraying attacks, its techniques can be applied to any attack that attempts to fill the address space with malicious code objects (e.g., stack spraying [42]).

1 Introduction

In recent years, security improvements have made it increasingly difficult for attackers to compromise systems. Successful prevention measures in runtime environments and operating systems include stack protection [10], improved heap allocation layouts [7, 20], address space layout randomization [8, 36], and data execution preven-

tion [21]. As a result, attacks that focus on exploiting memory corruptions in the heap are now popular [28].

Heap spraying, first described in 2004 by SkyLined [38], is an attack that allocates many objects containing the attacker's exploit code in an application's heap. Heap spraying is a vehicle for many high profile attacks, including a much publicized exploit in Internet Explorer in December 2008 [23] and a 2009 exploit of Adobe Reader using JavaScript embedded in malicious PDF documents [26].

Heap spraying requires that an attacker use another security exploit to trigger an attack, but the act of spraying greatly simplifies the attack and increases its likelihood of success because the exact addresses of objects in the heap do not need to be known. To perform heap spraying, attackers have to be able to allocate objects whose contents they control in an application's heap. The most common method used by attackers to achieve this goal is to target an application, such as a web browser, which executes an interpreter as part of its operation. By providing a web page with embedded JavaScript, an attacker can induce the interpreter to allocate their objects, allowing the spraying to occur. While this form of spraying attack is the most common, and the one we specifically consider in this paper, the techniques we describe apply to all forms of heap spraying. A number of variants of spraying attacks have recently been proposed including sprays involving compiled bytecode, ANI cursors [22], and thread stacks [42].

In this paper, we describe NOZZLE, a detector of heap spraying attacks that monitors heap activity and reports spraying attempts as they occur. To detect heap spraying attacks, NOZZLE has two complementary components. First, NOZZLE scans individual objects looking for signs of malicious intent. Malicious code commonly includes a landing pad of instructions (a so-called NOP sled) whose execution will lead to dangerous shellcode. NOZZLE focuses on detecting a sled through an analysis of its control flow. We show that prior work on sled detection [4, 16, 31, 43] has a high false positive rate when applied to objects in heap-spraying attacks (partly due to

the opcode density of the x86 instruction set). NOZZLE interprets individual objects as code and performs a static analysis, going beyond prior sled detection work by reasoning about code reachability. We define an attack surface metric that approximately answers the question: “If I were to jump randomly into this object (or heap), what is the likelihood that I would end up executing shellcode?”

In addition to local object detection, NOZZLE aggregates information about malicious objects across the entire heap, taking advantage of the fact that heap spraying requires large-scale changes to the contents of the heap. We develop a general notion of global “heap health” based on the measured attack surface of the application heap contents, and use this metric to reduce NOZZLE’s false positive rates.

Because NOZZLE only examines object contents and requires no changes to the object or heap structure, it can easily be integrated into both native and garbage-collected heaps. In this paper, we implement NOZZLE by intercepting calls to the memory manager in the Mozilla Firefox browser (version 2.0.0.16). Because browsers are the most popular target of heap spray attacks, it is crucial for a successful spray detector to both provide high successful detection rates and low false positive rates. While the focus of this paper is on low-overhead online detection of heap spraying, NOZZLE can be easily used for offline scanning to find malicious sites in the wild [45]. For offline scanning, we can combine our spraying detector with other checkers such as those that match signatures against the exploit code, etc.

1.1 Contributions

This paper makes the following contributions:

- We propose the first effective technique for detecting heap-spraying attacks through runtime interpretation and static analysis. We introduce the concept of attack surface area for both individual objects and the entire heap. Because directing program control to shellcode is a fundamental property of NOP sleds, the attacker cannot hide that intent from our analysis.
- We show that existing published sled detection techniques [4, 16, 31, 43] have high false positive rates when applied to heap objects. We describe new techniques that dramatically lower the false positive rate in this context.
- We measure Firefox interacting with popular web sites and published heap-spraying attacks, we show that NOZZLE successfully detects 100% of 12 published and 2,000 synthetically generated heap-spraying exploits. We also show that even with a detection threshold set six times lower than is required to detect known malicious attacks, NOZZLE

reports no false positives when tested on 150 popular Alexa.com sites.

- We measure the overhead of NOZZLE, showing that without sampling, examining every heap object slows execution 2–14 times. Using sampling and concurrent scanning, we show that the performance overhead of NOZZLE is less than 7% on average.
- We provide the results of applying NOZZLE to Adobe Reader to prevent a recent heap spraying exploit embedded in PDF documents. NOZZLE succeeds at stopping this attack without any modifications, with a runtime overhead of 8%.

1.2 Paper Organization

The rest of the paper is organized as follows. Section 2 provides background on heap spraying attacks. Section 3 provides an overview of NOZZLE and Section 4 goes into the technical details of our implementation. Section 5 summarizes our experimental results. While NOZZLE is the first published heap spraying detection technique, our approach has several limitations, which we describe fully in Section 6. Finally, Section 7 describes related work and Section 8 concludes.

2 Background

Heap spraying has much in common with existing stack and heap-based code injection attacks. In particular, the attacker attempts to inject code somewhere in the address space of the target program, and through a memory corruption exploit, coerce the program to jump to that code. Because the success of stack-based exploits has been reduced by the introduction of numerous security measures, heap-based attacks are now common. Injecting and exploiting code in the heap is more difficult for an attacker than placing code on the stack because the addresses of heap objects are less predictable than those of stack objects. Techniques such as address space layout randomization [8, 36] further reduce the predictability of objects on the heap. Attackers have adopted several strategies for overcoming this uncertainty [41], with heap spraying the most successful approach.

Figure 1 illustrates a common method of implementing a heap-spraying attack. Heap spraying requires a memory corruption exploit, as in our example, where an attacker has corrupted a vtable method pointer to point to an incorrect address of their choosing. At the same time, we assume that the attacker has been able, through entirely legal methods, to allocate objects with contents of their choosing on the heap. Heap spraying relies on populating the heap with a large number of objects containing the attacker’s code, assigning the vtable exploit to jump to an

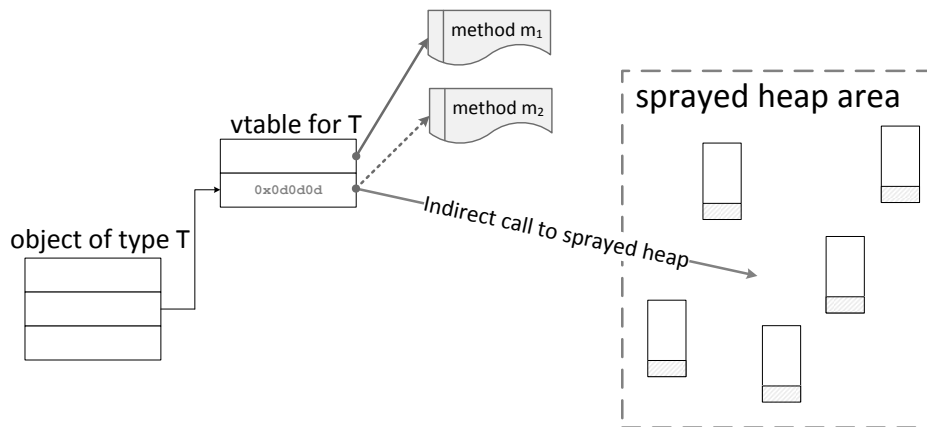


Figure 1: Schematic of a heap spraying attack.

```

1. <SCRIPT language="text/javascript">
2.   shellcode = unescape("%u4343%u4343%...");
3.   oneblock = unescape("%u0D0D%u0D0D");
4.
5.   var fullblock = oneblock;
6.   while (fullblock.length < 0x40000) {
7.     fullblock += oneblock;
8.   }
9.
10.  sprayContainer = new Array();
11.  for (i=0; i<1000; i++) {
12.    sprayContainer[i] = fullblock + shellcode;
13.  }
14. </SCRIPT>

```

Figure 2: A typical JavaScript heap spray.

arbitrary address in the heap, and relying on luck that the jump will land inside one of their objects. To increase the likelihood that the attack will succeed, attackers usually structure their objects to contain an initial NOP sled (indicated in white) followed by the code that implements the exploit (commonly referred to as shellcode, indicated with shading). Any jump that lands in the NOP sled will eventually transfer control to the shellcode. Increasing the size of the NOP sled and the number of sprayed objects increases the probability that the attack will be successful.

Heap spraying requires that the attacker control the contents of the heap in the process they are attacking. There are numerous ways to accomplish this goal, including providing data (such as a document or image) that when read into memory creates objects with the desired properties. An easier approach is to take advantage of scripting languages to allocate these objects directly. Browsers are particularly vulnerable to heap spraying because JavaScript embedded in a web page authored by the attacker greatly simplifies such attacks.

The example shown in Figure 2 is modelled after a previously published heap-spraying exploit [44]. While we

are only showing the JavaScript portion of the page, this payload would be typically embedded within an HTML page on the web. Once a victim visits the page, the JavaScript payload is automatically executed. Lines 2 allocates the shellcode into a string, while lines 3–8 of the JavaScript code are responsible for setting up the spraying NOP sled. Lines 10–13 create JavaScript objects each of which is the result of combining the sled with the shellcode. It is quite typical for published exploits to contain a long sled (256 KB in this case). Similarly, to increase the effectiveness of the attack, a large number of JavaScript objects are allocated on the heap, 1,000 in this case. Figure 10 in Section 5 provides more information on previously published exploits.

3 Overview

While type-safe languages such as Java, C#, and JavaScript reduce the opportunity for malicious attacks, heap-spraying attacks demonstrate that even a type-safe program can be manipulated to an attacker's advantage. Unfortunately, traditional signature-based pattern matching approaches used in the intrusion detection literature are not very effective when applied to detecting heap-spraying attacks. This is because in a language as flexible as JavaScript it is easy to hide the attack code by either using encodings or making it polymorphic; in fact, most JavaScript worms observed in the wild use some form of encoding to disguise themselves [19, 34]. As a result, effective detection techniques typically are not syntactic. They are performed at runtime and employ some level of semantic analysis or runtime interpretation. Hardware support has even been provided to address this problem, with widely used architectures supporting a “no-execute bit”, which prevents a process from executing code on specific pages in its address space [21]. We dis-

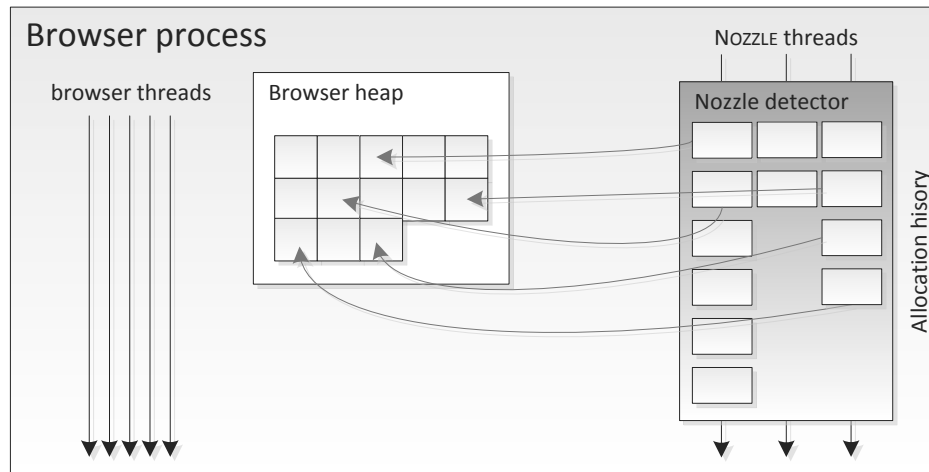


Figure 3: NOZZLE system architecture.

cuss how NOZZLE complements existing hardware solutions in Section 7. In this paper, we consider systems that use the x86 instruction set architecture (ISA) running the Windows operating system, a ubiquitous platform that is a popular target for attackers.

3.1 Lightweight Interpretation

Unlike previous security attacks, a successful heap-spraying attack has the property that the attack influences the contents of a large fraction of the heap. We propose a two-level approach to detecting such attacks: scanning objects locally while at the same time maintaining heap health metrics globally.

At the individual object level, NOZZLE performs lightweight interpretation of heap-allocated objects, treating them as though they were code. This allows us to recognize potentially unsafe code by interpreting it within a safe environment, looking for malicious intent.

The NOZZLE lightweight emulator scans heap objects to identify valid x86 code sequences, disassembling the code and building a control flow graph [35]. Our analysis focuses on detecting the NOP sled, which is somewhat of a misnomer. The sled can be composed of arbitrary instructions (not just NOPs) as long as the effect they have on registers, memory, and the rest of the machine state do not terminate execution or interfere with the actions of the shellcode. Because the code in the sled is intended to be the target of a misdirected jump, and thus has to be executable, the attacker cannot hide the sled with encryption or any means that would prevent the code from executing. In our analysis, we exploit the fundamental nature of the sled, which is to direct control flow specifically to the shellcode, and use this property as a means of detecting it. Furthermore, our method does not require detecting or

assume there exists a definite partition between the shellcode and the NOP sled.

Because the attack jump target cannot be precisely controlled, the emulator follows control flow to identify basic blocks that are likely to be reached through jumps from multiple offsets into the object. Our local detection process has elements in common with published methods for sled detection in network packet processing [4, 16, 31, 43]. Unfortunately, the density of the x86 instruction set makes the contents of many objects look like executable code, and as a result, published methods lead to high false positive rates, as demonstrated in Section 5.1.

We have developed a novel approach to mitigate this problem using global heap health metrics, which effectively distinguishes benign allocation behavior from malicious attacks. Fortunately, an inherent property of heap-spraying attacks is that such attacks affect the heap globally. Consequently, NOZZLE exploits this property to drastically reduce the false positive rate.

3.2 Threat Model

We assume that the attacker has access to memory vulnerabilities for commonly used browsers and also can lure users to a web site whose content they control. This provides a delivery mechanism for heap spraying exploits. We assume that the attacker does not have further access to the victim’s machine and the machine is otherwise uncompromised. However, the attacker does *not* control the precise location of any heap object.

We also assume that the attacker knows about the NOZZLE techniques and will try to avoid detection. They may have access to the browser code and possess detailed knowledge of system-specific memory layout properties

such as object alignment. There are specific potential weaknesses that NOZZLE has due to the nature of its runtime, statistical approach. These include time-of-check to time-of-use vulnerabilities, the ability of the attacker to target their attack under NOZZLE’s thresholds, and the approach of inserting junk bytes at the start of objects to avoid detection. We consider these vulnerabilities carefully in Section 6, after we have presented our solution in detail.

4 Design and Implementation

In this section, we formalize the problem of heap spray detection, provide improved algorithms for detecting suspicious heap objects, and describe the implementation of NOZZLE.

4.1 Formalization

This section formalizes our detection scheme informally described in Section 3.1, culminating in the notion of a *normalized attack surface*, a heap-global metric that reflects the overall heap exploitability and is used by NOZZLE to flag potential attacks.

Definition 1. A sequence of bytes is legitimate, if it can be decoded as a sequence of valid x86 instructions. In a variable length ISA this implies that the processor must be able to decode every instruction of the sequence. Specifically, for each instruction, the byte sequence consists of a valid opcode and the correct number of arguments for that instruction.

Unfortunately, the x86 instruction set is quite dense, and as a result, much of the heap data can be interpreted as legitimate x86 instructions. In our experiments, about 80% of objects allocated by Mozilla Firefox contain byte sequences that can be interpreted as x86 instructions.

Definition 2. A valid instruction sequence is a legitimate instruction sequence that does not include instructions in the following categories:

- I/O or system calls (`in`, `outs`, etc)
- interrupts (`int`)
- privileged instructions (`hlt`, `ltr`)
- jumps outside of the current object address range.

These instructions either divert control flow out of the object’s implied control flow graph or generate exceptions and terminate (privileged instructions). If they appear in a path of the NOP sled, they prevent control flow from reaching the shellcode via that path. When these instructions appear in the shellcode, they do not hamper the control flow in the NOP sled leading to that shellcode in any way.

Semi-lattice	L	bitvectors of length N
Top	\top	$\bar{1}$
Initial value	$init(B_i)$	$\bar{0}$
Transfer function	$TF(B_i)$	$0 \dots 010 \dots 0$ (i th bit set)
Meet operator	$\wedge(x, y)$	$x \vee y$ (bitwise or)
Direction		<i>forward</i>

Figure 4: Dataflow problem parametrization for computing the surface area (see Aho et al.).

Previous work on NOP sled detection focuses on examining possible attacks for properties like valid instruction sequences [4, 43]. We use this definition as a basic object filter, with results presented in Section 5.1. Using this approach as the sole technique for detecting attacks leads to an unacceptable number of false positives, and more selective techniques are necessary.

To improve our selectivity, NOZZLE attempts to discover objects in which control flow through the object (the NOP sled) frequently reaches the same basic block(s) (the shellcode, indicated in Figure 1), the assumption being that an attacker wants to arrange it so that a random jump into the object will reach the shellcode with the greatest probability.

Our algorithm constructs a control flow graph (CFG) by interpreting the data in an object at offset Δ as an instruction stream. For now, we consider this offset to be zero and discuss the implications of malicious code injected at a different starting offset in Section 6. As part of the construction process, we mark the basic blocks in the CFG as valid and invalid instruction sequences, and we modify the definition of a basic block so that it terminates when an invalid instruction is encountered. A block so terminated is considered an invalid instruction sequence. For every basic block within the CFG we compute the *surface area*, a proxy for the likelihood of control flow passing through the basic block, should the attacker jump to a random memory address within the object.

Algorithm 1. Surface area computation.

Inputs: Control flow graph C consisting of

- Basic blocks B_1, \dots, B_N
- Basic block weights, \bar{W} , a single-column vector of size N where element W_i indicates the size of block B_i in bytes
- A validity bitvector \bar{V} , a single-row bitvector whose i th element is set to one only when block B_i contains a valid instruction sequence and set to zero otherwise.
- $MASK_1, \dots, MASK_N$, where $MASK_i$ is a single-row bitvector of size N where all the bits are one except at the i^{th} position where the bit is zero.

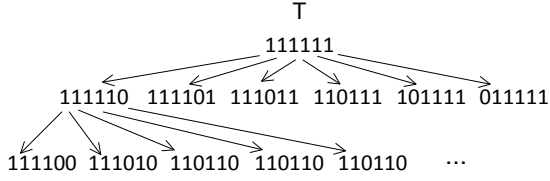


Figure 5: Semi-lattice used in Example 1.

Outputs: Surface area for each basic block $SA(B_i)$, $B_i \in C$.

Solution: We define a parameterized dataflow problem using the terminology in Aho et al. [2], as shown in Figure 4. We also relax the definition of a conventional basic block; whenever an invalid instruction is encountered, the block prematurely terminates. The goal of the dataflow analysis is to compute the reachability between basic blocks in the control graph inferred from the contents of the object. Specifically, we want to determine whether control flow could possibly pass through a given basic block if control starts at each of the other $N - 1$ blocks. Intuitively, if control reaches a basic block from many of the other blocks in the object (demonstrating a “funnel” effect), then that object exhibits behavior consistent with having a NOP sled and is suspicious.

Dataflow analysis details: The dataflow solution computes $out(B_i)$ for every basic block $B_i \in C$. $out(B_i)$ is a bitvector of length N , with one bit for each basic block in the control flow graph. The meaning of the bits in $out(B_i)$ are as follows: the bit at position j , where $j \neq i$ indicates whether a possible control path exists starting at block j and ending at block i . The bit at position i in B_i is always one. For example, in Figure 6, a path exists between block 1 and 2 (a fallthrough), and so the first bit of $out(B_2)$ is set to 1. Likewise, there is no path from block 6 to block 1, so the sixth bit of $out(B_1)$ is zero.

The dataflow algorithm computes $out(B_i)$ for each B_i by initializing them, computing the contribution that each basic block makes to $out(B_i)$, and propagating intermediate results from each basic block to its successors (because this is a forward dataflow computation). When results from two predecessors need to be combined at a join point, the meet operator is used (in this case a simple bitwise or). The dataflow algorithm iterates the forward propagation until the results computed for each B_i do not change further. When no further changes occur, the final values of $out(B_i)$ have been computed. The iterative algorithm for this forward dataflow problem is guaranteed to terminate in no more than the number of steps equal to the product of the semi-lattice height and the number of basic blocks in the control flow graph [2].

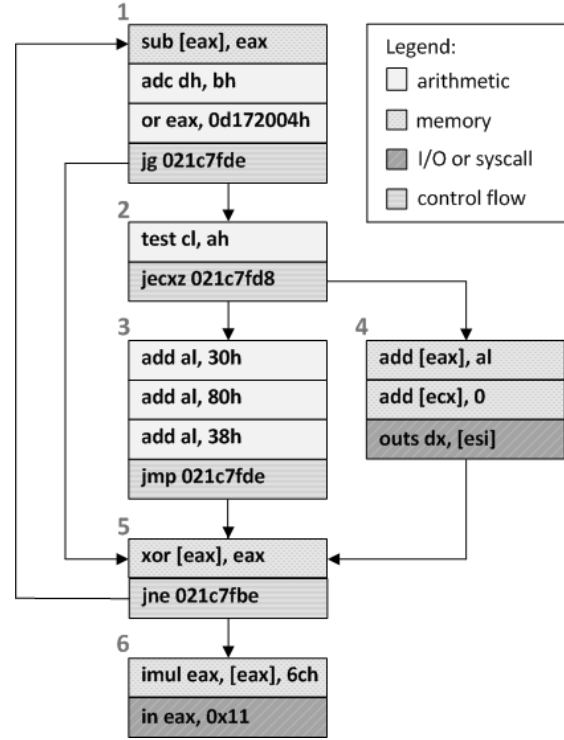


Figure 6: The control flow graph for Example 1.

Having calculated $out(B_i)$, we are now ready to compute the surface area of the basic block B_i . The surface area of a given block is a metric that indicates how likely the block will be reached given a random control flow landing on this object. The surface area of basic block B_i , $SA(B_i)$, is computed as follows:

$$SA(B_i) = (out(B_i) \wedge \bar{V} \wedge MASK_i) \cdot \bar{W}$$

where $out(B_i)$ is represented by a bitvector whose values are computed using the iterative dataflow algorithm above. \bar{V} , \bar{W} , and $MASK_i$ are the algorithm’s inputs. \bar{V} is determined using the validity criteria mentioned above, while \bar{W} is the size of each basic block in bytes. $MASK_i$ is used to mask out the contribution of B_i ’s weight to its own surface area. The intuition is that we discard the contribution from the block itself as well as other basic blocks that are not valid instruction sequences by logically bit-wise ANDing $out(B_i)$, \bar{V} , and $MASK_i$. Because the shellcode block does not contribute to actual attack surface (since a jump inside the shellcode is not likely to result in a successful exploit), we do not include the weight of B_i as part of the attack surface. Finally, we perform vector multiplication to account for the weight each basic block contributes—or does not—to the surface area of B_i .

In summary, the surface area computation based on the dataflow framework we described accounts for the contribution each basic block, through its weight and validity,

has on every other blocks reachable by it. Our computation method can handle code with complex control flow involving arbitrary nested loops. It also allows for the discovery of malicious objects even if the object has no clear partition between the NOP sled and the shellcode itself.

Complexity analysis. The standard iterative algorithm for solving dataflow problems computes $out(B_i)$ values with an average complexity bound of $O(N)$. The only complication is that doing the lattice meet operation on bitvectors of length N is generally an $O(N)$ and *not* a constant time operation. Luckily, for the majority of CFGs that arise in practice — 99.08% in the case of Mozilla Firefox opened and interacted on `www.google.com` — the number of basic blocks is fewer than 64, which allows us to represent dataflow values as long integers on 64-bit hardware. For those rare CFGs that contain over 64 basic blocks, a generic bitvector implementation is needed.

Example 1 Consider the CFG in Figure 6. The semi-lattice for this CFG of size 6 is partially shown in Figure 5. Instructions in the CFG are color-coded by instruction type. In particular, system calls and I/O instructions interrupt the normal control flow. For simplicity, we show \bar{W}_i as the number of instructions in each block, instead of the number of bytes. The values used and produced by the algorithm are summarized in Figure 7. The $out'(B_i)$ column shows the intermediate results for dataflow calculation after the first pass. The final solution is shown in the $out(B_i)$ column. \square

Given the surface area of individual blocks, we compute the *attack surface area* of object o as:

$$SA(o) = \max(SA(B_i), B_i \in C)$$

For the entire heap, we accumulate the attack surface of the individual objects.

Definition 3. The attack surface area of heap H , $SA(H)$, containing objects o_1, \dots, o_n is defined as follows:

$$\sum_{i=1, \dots, n} SA(o_i)$$

Definition 4. The normalized attack surface area of heap H , denoted as $NSA(H)$, is defined as: $SA(H)/|H|$.

The normalized attack surface area metric reflects the overall heap “health” and also allows us to adjust the frequency with which NOZZLE runs, thereby reducing the runtime overhead, as explained below.

4.2 Nozzle Implementation

NOZZLE needs to periodically scan heap object content in a way that is analogous to a garbage collector mark phase.

By instrumenting allocation and deallocation routines, we maintain a table of live objects that are later scanned asynchronously, on a different NOZZLE thread.

We adopt garbage collection terminology in our description because the techniques are similar. For example, we refer to the threads allocating and freeing objects as the mutator threads, while we call the NOZZLE threads scanning threads. While there are similarities, there are also key differences. For example, NOZZLE works on an unmanaged, type-unsafe heap. If we had garbage collector write barriers, it would improve our ability to address the TOCTTOU (time-of-check to time-of-use) issue discussed in Section 6.

4.2.1 Detouring Memory Management Routines

We use a binary rewriting infrastructure called Detours [14] to intercept functions calls that allocate and free memory. Within Mozilla Firefox these routines are `malloc`, `calloc`, `realloc`, and `free`, defined in `MOZCRT19.dll`. To compute the surface area, we maintain information about the heap including the total size of allocated objects.

NOZZLE maintains a hash table that maps the addresses of currently allocated objects to information including size, which is used to track the current size and contents of the heap. When objects are freed, we remove them from the hash table and update the size of the heap accordingly. Note that if NOZZLE were more closely integrated into the heap allocator itself, this hash table would be unnecessary.

NOZZLE maintains an ordered work queue that serves two purposes. First, it is used by the scanning thread as a source of objects that need to be scanned. Second, NOZZLE waits for objects to mature before they are scanned, and this queue serves that purpose. Nozzle only considers objects of size greater than 32 bytes to be put in the work queue as the size of any harmful shellcode is usually larger than this.

To reduce the runtime overhead of NOZZLE, we randomly sample a subset of heap objects, with the goal of covering a fixed fraction of the total heap. Our current sampling technique is based on sampling by object, but as our results show, an improved technique would base sampling frequency on bytes allocated, as some of the published attacks allocate a relatively small number of large objects.

4.2.2 Concurrent Object Scanning

We can reduce the performance impact of object scanning, especially on multicore hardware, with the help of multiple scanning threads. As part of program detouring, we rewrite the `main` function to allocate a pool of N scanning threads to be used by NOZZLE, as shown in Figure 2.

B_i	$TF(B_i)$	\bar{V}_i	\bar{W}_i	$out'(B_i)$	$out(B_i)$	$out(B_i) \wedge \bar{V} \wedge MASK_i$	$SA(B_i)$
1	100000	1	4	100000	111110	011010	8
2	010000	1	2	110000	111110	101010	10
3	001000	1	4	111000	111110	110010	8
4	000100	0	3	110100	111110	111010	12
5	000010	1	2	111110	111110	111000	10
6	000001	0	2	111111	111111	111010	12

Figure 7: Dataflow values for Example 1.

This way, a mutator only blocks long enough when allocating and freeing objects to add or remove objects from a per-thread work queue.

The task of object scanning is subdivided among the scanning threads the following way: for an object at address a , thread number

$$(a \gg p) \% N$$

is responsible for both maintaining information about that object and scanning it, where p is the number of bits required to encode the operating system page size (typically 12 on Windows). In other words, to preserve the spatial locality of heap access, we are distributing the task of scanning *individual pages* among the N threads. Instead of maintaining a global hash table, each thread maintains a local table keeping track of the sizes for the objects it handles.

Object scanning can be triggered by a variety of events. Our current implementation scans objects once, after a fixed delay of one object allocation (i.e., we scan the previously allocated object when we see the next object allocated). This choice works well for JavaScript, where string objects are immutable, and hence initialized immediately after they are allocated. Alternately, if there are extra cores available, scanning threads could pro-actively rescan objects without impacting browser performance and reducing TOCTTOU vulnerabilities (see Section 6).

4.3 Detection and Reporting

NOZZLE maintains the values $NSA(H)$ and $SA(H)$ for the currently allocated heap H . The criteria we use to conclude that there is an attack in progress combines an absolute and a relative threshold:

$$(NSA(H) > th_{norm}) \wedge (SA(H) > th_{abs})$$

When this condition is satisfied, we warn the user about a potential security attack in progress and allow them to kill the browser process. An alternative would be to take advantage of the error reporting infrastructure built into modern browsers to notify the browser vendor.

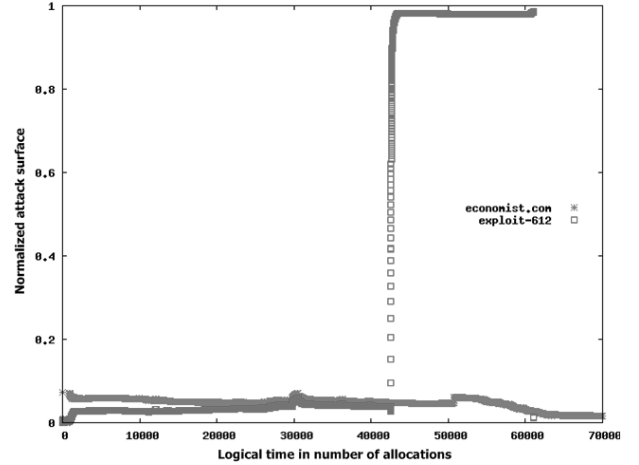


Figure 8: Global normalized attack surface for economist.com versus a published exploit (612).

These thresholds are defined based on a comparison of benign and malicious web pages (Section 5.1). The guiding principle behind the threshold determination is that for the attacker to succeed, the exploit needs to be effective with reasonable probability. For the absolute threshold, we choose five megabytes, which is roughly the size of the Firefox heap when opening a blank page. A real attack would need to fill the heap with at least as many malicious objects, assuming the attacker wanted the ratio of malicious to non-malicious objects to be greater than 50%.

5 Evaluation

The bulk of our evaluation focuses on applying NOZZLE to the Firefox web browser. Section 5.5 talks about using NOZZLE to protect Adobe Acrobat Reader.

We begin our evaluation by showing what a heap-spraying attack looks like as measured using our normalized attack surface metric. Figure 8 shows the attack surface area of the heap for two web sites: a benign site (economist.com), and a site with a published heap-spraying attack, similar to the one presented in Figure 2. Figure 8 illustrates how distinctive a heap-spraying attack

is when viewed through the normalized attack surface filter. The success of NOZZLE depends on its ability to distinguish between these two kinds of behavior. After seeing Figure 8, one might conclude that we can detect heap spraying activity based on how rapidly the heap grows. Unfortunately, benign web sites as `economist.com` can possess as high a heap growth rate as a rogue page performing heap spraying. Moreover, unhurried attackers may avoid such detection by moderating the heap growth rate of their spray. In this section, we present the false positive and false negative rate of NOZZLE, as well as its performance overhead, demonstrating that it can effectively distinguish benign from malicious sites.

For our evaluations, we collected 10 heavily-used benign web sites with a variety of content and levels of scripting, which we summarize in Figure 9. We use these 10 sites to measure the false positive rate and also the impact of NOZZLE on browser performance, discussed in Section 5.3. In our measurements, when visiting these sites, we interacted with the site as a normal user would, finding a location on a map, requesting driving directions, etc. Because such interaction is hard to script and reproduce, we also studied the false positive rate of NOZZLE using a total of 150 benign web sites, chosen from the most visited sites as ranked by Alexa [5]¹. For these sites, we simply loaded the first page of the site and measured the heap activity caused by that page alone.

To evaluate NOZZLE’s ability to detect malicious attacks, we gathered 12 published heap-spraying exploits, summarized in Figure 10. We also created 2,000 synthetically generated exploits using the Metasploit framework [12]. Metasploit allows us to create many malicious code sequences with a wide variety of NOP sled and shellcode contents, so that we can evaluate the ability of our algorithms to detect such attacks. Metasploit is parameterizable, and as a result, we can create attacks that contain NOP sleds alone, or NOP sleds plus shellcode. In creating our Metasploit exploits, we set the ratio of NOP sled to shellcode at 9:1, which is quite a low ratio for a real attack but nevertheless presents no problems for NOZZLE detection.

5.1 False Positives

To evaluate the false positive rate, we first consider using NOZZLE as a global detector determining whether a heap is under attack, and then consider the false-positive rate of NOZZLE as a local detector that is attempting to detect individual malicious objects. In our evaluation, we compare NOZZLE and STRIDE [4], a recently published local detector.

¹Our tech report lists the full set of sites used [32].

Site URL	Download (kilobytes)	JavaScript (kilobytes)	Load time (seconds)
<code>economist.com</code>	613	112	12.6
<code>cnn.com</code>	885	299	22.6
<code>yahoo.com</code>	268	145	6.6
<code>google.com</code>	25	0	0.9
<code>amazon.com</code>	500	22	14.8
<code>ebay.com</code>	362	52	5.5
<code>facebook.com</code>	77	22	4.9
<code>youtube.com</code>	820	160	16.5
<code>maps.google.com</code>	285	0	14.2
<code>maps.live.com</code>	3000	2000	13.6

Figure 9: Summary of 10 benign web sites we used as NOZZLE benchmarks.

Date	Browser	Description	milw0rm
11/2004	IE	IFRAME Tag BO	612
04/2005	IE	DHTML Objects Corruption	930
01/2005	IE	.ANI Remote Stack BO	753
07/2005	IE	javaprx.dll COM Object	1079
03/2006	IE	createTextRange RE	1606
09/2006	IE	VML Remote BO	2408
03/2007	IE	ADODB Double Free	3577
09/2006	IE	WebViewFolderIcon setSlice	2448
09/2005	FF	0xAD Remote Heap BO	1224
12/2005	FF	compareTo() RE	1369
07/2006	FF	Navigator Object RE	2082
07/2008	Safari	Quicktime Content-Type BO	6013

Figure 10: Summary of information about 12 published heap-spraying exploits. BO stands for “buffer overruns” and RE stands for “remote execution.”

5.1.1 Global False Positive Rate

Figure 11 shows the maximum normalized attack surface measured by NOZZLE for our 10 benchmark sites (top) as well as the top 150 sites reported by Alexa (bottom). From the figure, we see that the maximum normalized attack surface remains around 6% for most of the sites, with a single outlier from the 150 sites at 12%. In practice, the median attack surface is typically much lower than this, with the maximum often occurring early in the rendering of the page when the heap is relatively small. The `economist.com` line in Figure 8 illustrates this effect. By setting the spray detection threshold at 15% or above, we would observe no false positives in any of the sites measured.

5.1.2 Local False Positive Rate

In addition to being used as a heap-spray detector, NOZZLE can also be used locally as a malicious object detector. In this use, as with existing NOP and shellcode detectors such as STRIDE [4], a tool would report an object as potentially malicious if it contained data that could be interpreted as code, and had other suspicious properties. Previous work in this area focused on detection of malware in network packets and URIs, whose content is very different than heap objects. We evaluated NOZZLE

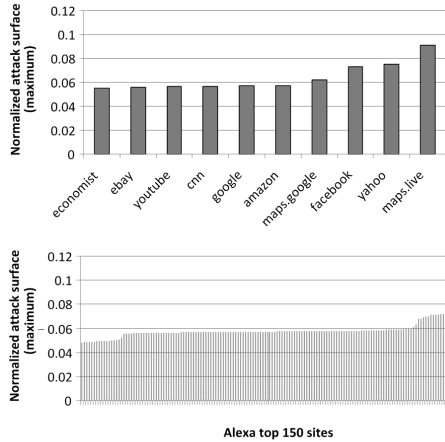


Figure 11: Global normalized attack surface for 10 benign benchmark web sites and 150 additional top Alexa sites, sorted by increasing surface. Each element of the X-axis represents a different web site.

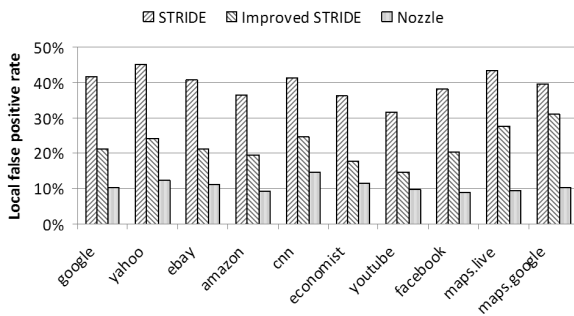


Figure 12: Local false positive rate for 10 benchmark web sites using NOZZLE and STRIDE. Improved STRIDE is a version of STRIDE that uses additional instruction-level filters, also used in NOZZLE, to reduce the false positive rate.

and STRIDE algorithm, to see how effective they are at classifying benign heap objects.

Figure 12 indicates the false positive rate of two variants of STRIDE and a simplified variant of NOZZLE. This simplified version of NOZZLE only scans a given heap object and attempts to disassemble and build a control flow graph from its contents. If it succeeds in doing this, it considers the object suspect. This version does not include any attack surface computation. The figure shows that, unlike previously reported work where the false positive rate for URIs was extremely low, the false positive rate for heap objects is quite high, sometimes above 40%. An improved variant of STRIDE that uses more information about the x86 instruction set (also used in NOZZLE) reduces this rate, but not below 10% in any case. We con-

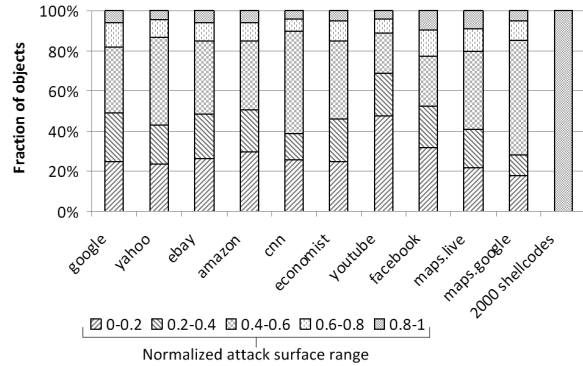


Figure 13: Distribution of filtered object surface area for each of 10 benchmark web sites (benign) plus 2,000 synthetic exploits (see Section 5.2). Objects measured are only those that were considered valid instruction sequences by NOZZLE (indicated as false positives in Figure 12).

clude that, unlike URIs or the content of network packets, heap objects often have contents that can be entirely interpreted as code on the x86 architecture. As a result, existing methods of sled detection do not directly apply to heap objects. We also show that even NOZZLE, without incorporating our surface area computation, would have an unacceptably high false positive rate.

To increase the precision of a local detector based on NOZZLE, we incorporate the surface area calculation described in Section 4. Figure 13 indicates the distribution of measured surface areas for the roughly 10% of objects in Figure 12 that our simplified version of NOZZLE was not able to filter. We see from the figure that many of those objects have a relatively small surface area, with less than 10% having surface areas from 80-100% of the size of the object (the top part of each bar). Thus, roughly 1% of objects allocated by our benchmark web sites qualify as suspicious by a local NOZZLE detector, compared to roughly 20% using methods reported in prior work. Even at 1%, the false positive rate of a local NOZZLE detector is too high to raise an alarm whenever a single instance of a suspicious object is observed, which motivated the development of our global heap health metric.

5.2 False Negatives

As with the false positive evaluation, we can consider NOZZLE both as a local detector (evaluating if NOZZLE is capable of classifying a known malicious object correctly), and as a global detector, evaluating whether it correctly detects web pages that attempt to spray many copies of malicious objects in the heap.

Figure 14 evaluates how effective NOZZLE is at avoid-

Configuration	min	mean	std
Local, NOP w/o shellcode	0.994	0.997	0.002
Local, NOP with shellcode	0.902	0.949	0.027

Figure 14: Local attack surface metrics for 2,000 generated samples from Metasploit with and without shellcode.

Configuration	min	mean	std
Global, published exploits	0.892	0.966	0.028
Global, Metasploit exploits	0.729	0.760	0.016

Figure 15: Global attack surface metrics for 12 published attacks and 2,000 Metasploit attacks integrated into web pages as heap sprays.

ing local false negatives using our Metasploit exploits. The figure indicates the mean and standard deviation of the object surface area over the collection of 2,000 exploits, both when shellcode is included with the NOP sled and when the NOP sled is measured alone. The figure shows that NOZZLE computes a very high attack surface in both cases, effectively detecting all the Metasploit exploits both with and without shellcode.

Figure 15 shows the attack surface statistics when using NOZZLE as a global detector when the real and synthetic exploits are embedded into a web page as a heap-spraying attack. For the Metasploit exploits which were not specifically generated to be heap-spraying attacks, we wrote our own JavaScript code to spray the objects in the heap. The figure shows that the published exploits are more aggressive than our synthetic exploits, resulting in a mean global attack surface of 97%. For our synthetic use of spraying, which was more conservative, we still measured a mean global attack surface of 76%. Note that if we set the NOP sled to shellcode at a ratio lower than 9:1, we will observe a correspondingly smaller value for the mean global attack surface. All attacks would be detected by NOZZLE with a relatively modest threshold setting of 50%. We note that these exploits have global attack surface metrics 6–8 times larger than the maximum measured attack surface of a benign web site.

5.3 Performance

To measure the performance overhead of NOZZLE, we cached a typical page for each of our 10 benchmark sites. We then instrument the start and the end of the page with the JavaScript `newDate().getTime()` routine and compute the delta between the two. This value gives us how long it takes to load a page in milliseconds. We collect our measurements running Firefox version 2.0.0.16 on a 2.4 GHz Intel Core2 E6600 CPU running Windows XP

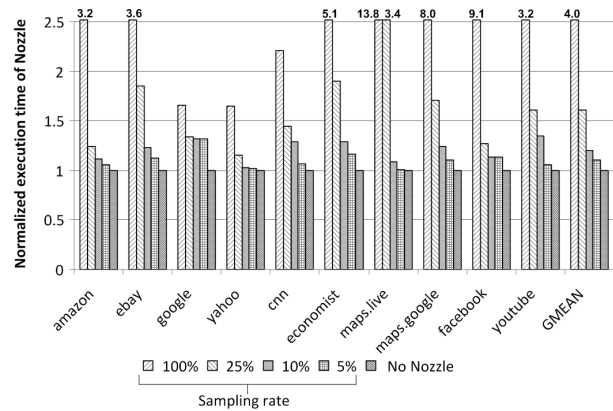


Figure 16: Relative execution overhead of using NOZZLE in rendering a typical page of 10 benchmark web sites as a function of sampling frequency.

Service Pack 3 with 2 gigabytes of main memory. To minimize the effect of timing due to cold start disk I/O, we first load a page and discard the timing measurement. After this first trial, we take three measurements and present the median of the three values. The experiments were performed on an otherwise quiescent machine and the variance between runs was not significant.

In the first measurement, we measured the overhead of NOZZLE without leveraging an additional core, i.e., running NOZZLE as a single thread and, hence, blocking Firefox every time a memory allocation occurs. The resulting overhead is shown in Figure 16, both with and without sampling. The overhead is prohibitively large when no sampling is applied. On average, the no sampling approach incurs about 4X slowdown with as much as 13X slowdown for `maps.live.com`. To reduce this overhead, we consider the sampling approach. For these results, we sample based on object counts; for example, sampling at 5% indicates that one in twenty objects is sampled. Because a heap-spraying attack has global impact on the heap, sampling is unlikely to significantly reduce our false positive and false negative rates, as we show in the next section. As we reduce the sampling frequency, the overhead becomes more manageable. We see an average slowdown of about 60%, 20% and 10% for sampling frequency of 25%, 10% and 5%, respectively, for the 10 selected sites.

For the second measurement, taking advantage of the second core of our dual core machine, we configured NOZZLE to use one additional thread for scanning, hence, unblocking Firefox when it performs memory allocation. Figure 17 shows the performance overhead of NOZZLE with parallel scanning. From the Figure, we see that with no sampling, the overhead of using NOZZLE ranges from 30% to almost a factor of six, with a geometric mean of

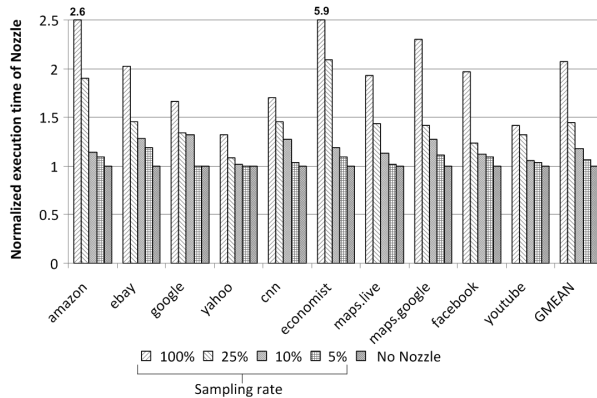


Figure 17: Overhead of using NOZZLE on a dual-core machine.

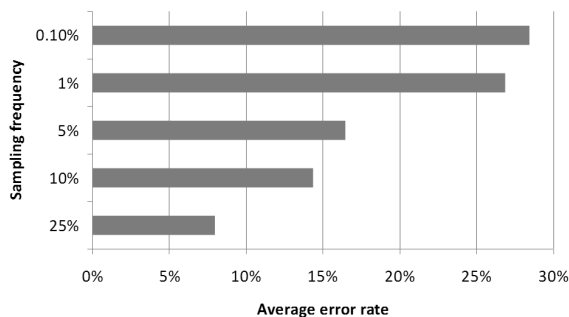


Figure 18: Average error rate due to sampling of the computed average surface area for 10 benign benchmark web sites.

two times slowdown. This is a significant improvement over the serial version. When we further reduce the sampling rate, we see further performance improvement as with the first measurement. Reducing the sampling rate to 25%, the mean overhead drops to 45%, while with a sampling rate of 5%, the performance overhead is only 6.4%.

5.4 Impact of Sampling on Detection

In this section, we show the impact of sampling on the amount of error in the computation of the attack surface metric for both benign and malicious inputs.

Figure 18 shows the error rate caused by different levels of sampling averaged across the 10 benign web sites. We compute the error rate $E = |Sampled - Unsampled| / Unsampled$. The figure shows that for sample rates of 0.1% or above the error rate is less than 30%. To make this concrete, for a benign website, instead of calculating the normalized attack surface correctly as 5%, with a 0.1% sampling rate, we would instead calcu-

	Sampling Rate				
	100%	25%	10%	5%	1%
12 Published	0	0	0	0	50%
2,000 Metasploit	0	0	0	0	0

Figure 19: False negative rate for 12 real and 2,000 Metasploit attacks given different object sampling rates.

late the normalized attack surface as 6.5%, still far below any threshold we might use for signaling an attack. Noting that the malicious pages have attack surfaces that are 6–8 times larger than benign web pages, we conclude that sampling even at 5% is unlikely to result in significant numbers of false positives.

In Figure 19, we show the impact of sampling on the number of false negatives for our published and synthetic exploits. Because existing exploits involve generating the heap spray in a loop, the only way sampling will miss such an attack is to sample at such a low rate that the objects allocated in the loop escape notice. The figure illustrates that for published attacks sampling even at 5% results in no false negatives. At 1%, because several of the published exploits only create on the order of tens of copies of very large spray objects, sampling based on object count can miss these objects, and we observe a 50% (6/12) false negative rate. Sampling based on bytes allocated instead of objects allocated would reduce this false negative rate to zero.

5.5 Case Study: Adobe Reader Exploit

In February 2009, a remote code execution vulnerability was discovered in Adobe Acrobat and Adobe Reader [26]. The attack, which is still active on the Internet as of the time of this writing, exploited an integer overflow error and was facilitated by a JavaScript heap spray. Without making any modifications to NOZZLE, we used Detours to instrument the commercially-distributed binary of Adobe Reader 9.1.0 (acrord32.exe) with NOZZLE. The instrumentation allowed us to monitor the memory allocations being performed by the embedded JavaScript engine and detect possible spraying attacks. To test whether NOZZLE would detect this new attack, we embedded the heap spraying part of the published attack [6], disabling the JavaScript that caused the integer overflow exploit.

NOZZLE correctly detected this heap spraying attack, determining that the attack surface of the heap was greater than 94% by the time the heap spray was finished. No modifications were made either to the NOZZLE implementation or the surface area calculation to enable NOZZLE to detect this attack, which gives us confidence that NOZZLE is capable of protecting a wide range of software, going well beyond just web browsers.

To facilitate overhead measurements, we created a large document by concatenating six copies of the ECMA 262 standard — a 188-page PDF document [11] — with itself. The resulting document was 1,128 pages long and took 4,207 kilobytes of disk space. We added scripting code to the document to force Adobe Reader to “scroll” through this large document, rendering every page sequentially. We believe this workload to be representative of typical Adobe Reader usage, where the user pages through the document, one page at a time.

We measured the overhead of NOZZLE running in Adobe Reader on an Intel Core 2 2.4 GHz computer with 4 GB of memory running Windows Vista SP1. We measured elapsed time for Adobe Reader with and without NOZZLE on a lightly loaded computer and averaged five measurements with little observed variation. Without NOZZLE, Adobe Reader took an average of 18.7 seconds to render all the pages, and had a private working set of 18,772 kilobytes as measured with the Windows Task Manager. With a sampling rate set to 10% and multiprocessor scanning disabled, Adobe Reader with NOZZLE took an average of 20.2 seconds to render the pages, an average CPU overhead of 8%. The working set of Adobe Reader with NOZZLE average 31,849 kilobytes, an average memory overhead of 69%. While the memory overhead is high, as mentioned, we anticipate that this overhead could easily be reduced by integrating NOZZLE more closely with the underlying memory manager.

6 Limitations of NOZZLE

This section discusses assumptions and limitations of the current version of NOZZLE. In summary, assuming that the attackers are fully aware of the NOZZLE internals, there are a number of ways to evade its detection.

- As NOZZLE scans objects at specific times, an attacker could determine that an object has been scanned and arrange to put malicious content into the object only *after* it has been scanned (a TOCTTOU vulnerability).
- As NOZZLE currently starts scanning each object at offset zero, attackers can avoid detection by writing the first few bytes of the malicious object with a series of uninterpretable opcodes.
- Since NOZZLE relies on the use of a threshold for detection, attackers can populate the heap with fewer malicious objects to stay just under the detection threshold.
- Attackers can find ways to inject the heap with sprays that do not require large NOP sleds. For example, sprays with jump targets that are at fixed offsets in every sprayed page of memory are possible [39].

- Attackers can confuse NOZZLE’s surface area measurement by designing attacks that embed multiple shellcodes within the same object or contain cross-object jumps.

Below we discuss these issues, their implications, and possible ways to address them.

6.1 Time-of-check to Time-of-use

Because NOZZLE examines object contents only at specific times, this leads to a potential time-of-check to time-of-use (TOCTTOU) vulnerability. An attacker aware that NOZZLE was being used could allocate a benign object, wait until NOZZLE scans it, and then rapidly change the object into a malicious one before executing the attack.

With JavaScript-based attacks, since JavaScript Strings are immutable, this is generally only possible using JavaScript Arrays. Further, because NOZZLE may not know when objects are completely initialized, it may scan them prematurely, creating another TOCTTOU window. To address this issue, NOZZLE scans objects once they mature on the assumption that most objects are written once when initialized, soon after they are allocated. In the future, we intend to investigate other ways to reduce this vulnerability, including periodically rescanning objects. Rescans could be triggered when NOZZLE observes a significant number of heap stores, perhaps by reading hardware performance counters.

Moreover, in the case of a garbage-collected language such as JavaScript or Java, NOZZLE can be integrated directly with the garbage collector. In this case, changes observed via GC *write barriers* [29] may be used to trigger NOZZLE scanning.

6.2 Interpretation Start Offset

In our discussion so far, we have interpreted the contents of objects as instructions starting at offset zero in the object, which allows NOZZLE to detect the current generation of heap-spraying exploits. However, if attackers are aware that NOZZLE is being used, they could arrange to fool NOZZLE by inserting junk bytes at the start of objects. There are several reasons that such techniques will not be as successful as one might think. To counter the most simplistic such attack, if there are invalid or illegal instructions at the beginning of the object, NOZZLE skips bytes until it finds the first valid instruction.

Note that while it may seem that the attacker has much flexibility to engineer the offset of the start of the malicious code, the attacker is constrained in several important ways. First, we know that it is likely that the attacker is trying to maximize the probability that the attack will succeed. Second, recall that the attacker has to corrupt a control transfer but does not know the specific address in an

object where the jump will land. If the jump lands on an invalid or illegal instruction, then the attack will fail. As a result, the attacker may seek to make a control transfer to every address in the malicious object result in an exploit. If this is the case, then NOZZLE will correctly detect the malicious code. Finally, if the attacker knows that NOZZLE will start interpreting the data as instructions starting at a particular offset, then the attacker might intentionally construct the sled in such a way that the induced instructions starting at one offset look benign, while the induced instructions starting at a different offset are malicious. For example, the simplest form of this kind of attack would have uniform 4-byte benign instructions starting at byte offset 0 and uniform malicious 4-byte instructions starting at byte offset 2. Note also that these overlapped sequences cannot share any instruction boundaries because if they did, then processing instructions starting at the benign offset would eventually discover malicious instructions at the point where the two sequences merged.

While the current version of NOZZLE does not address this specific simple case, NOZZLE could be modified to handle it by generating multiple control flow graphs at multiple starting offsets. Furthermore, because x86 instructions are typically short, most induced instruction sequences starting at different offsets do not have many possible interpretations before they share a common instruction boundary and merge. While it is theoretically possible for a determined attacker to create a non-regular, non-overlapping sequence of benign and malicious instructions, it is not obvious that the malicious sequence could not be discovered by performing object scans at a small number of offsets into the object. We leave an analysis of such techniques for future work.

6.3 Threshold Setting

The success of heap spraying is directly proportional to the density of dangerous objects in the program heap, which is approximated by NOZZLE's normalized attack surface metric. Increasing the number of sprayed malicious objects increases the attacker's likelihood of success, but at the same time, more sprayed objects will increase the likelihood that NOZZLE will detect the attack. What is worse for the attacker, failing attacks often result in program crashes. In the browser context, these are recorded on the user's machine and sent to browser vendors using a crash agent such as Mozilla Crash reporting [24] for per-site bucketing and analysis.

What is interesting about attacks against browsers is that from a purely financial standpoint, the attacker has a strong incentive to produce exploits with a high likelihood of success. Indeed, assuming the attacker is the one discovering the vulnerability such as a dangling pointer or buffer overrun enabling the heap-spraying attack, they

can sell their finding directly to the browser maker. For instance, the Mozilla Foundation, the makers of Firefox, offers a cash reward of \$500 for every exploitable vulnerability [25]. This represents a conservative estimate of the financial value of such an exploit, given that Mozilla is a non-profit and commercial browser makers are likely to pay more [15]. A key realization is that in many cases heap spraying is used for direct financial gain. A typical way to monetize a heap-spraying attack is to take over a number of unsuspecting users' computers and have them join a botnet. A large-scale botnet can be sold on the black market to be used for spamming or DDOS attacks.

According to some reports, the cost of a large-scale botnet is about \$.10 per machine [40, 18]. So, to break even, the attacker has to take over at least 5,000 computers. For a success rate α , in addition to 5,000 successfully compromised machines, there are $5,000 \times (1 - \alpha)/\alpha$ failed attacks. Even a success rate as high as 90%, the attack campaign will produce failures for 555 users. Assuming these result in crashes reported by the crash agent, this many reports from a single web site may attract attention of the browser maker. For a success rate of 50%, the browser maker is likely to receive 5,000 crash reports, which should lead to rapid detection of the exploit!

As discussed in Section 5, in practice we use the relative threshold of 50% with Nozzle. We do not believe that a much lower success rate is financially viable from the standpoint of the attacker.

6.4 Targeted Jumps into Pages

One approach to circumventing NOZZLE detection is for the attacker to eliminate the large NOP sled that heap sprays typically use. This may be accomplished by allocating page-size chunks of memory (or multiples thereof) and placing the shellcode at fixed offsets on every page [39]. While our spraying detection technique currently will not discover such attacks, it is possible that the presence of possible shellcode at fixed offsets on a large number of user-allocated pages can be detected by extending NOZZLE, which we will consider in future work.

6.5 Confusing Control Flow Patterns

NOZZLE attempts to find basic blocks that act as sinks for random jumps into objects. One approach that will confuse NOZZLE is to include a large number of copies of shellcode in an object such that no one of them has a high surface area. Such an approach would still require that a high percentage of random jumps into objects result in non-terminating control flow, which might also be used as a trigger for our detector.

Even more problematic is an attack where the attacker includes inter-object jumps, under the assumption that,

probabilistically, there will be a high density of malicious objects and hence jumps outside of the current object will still land in another malicious object. NOZZLE currently assumes that jumps outside of the current object will result in termination. We anticipate that our control flow analysis can be augmented to detect groupings of objects with possible inter-object control flow, but we leave this problem for future work.

6.6 Summary

In summary, there are a number of ways that clever attackers can defeat NOZZLE's current analysis techniques. Nevertheless, we consider NOZZLE an important first step to detecting heap spraying attacks and we believe that improvements to our techniques are possible and will be implemented, just as attackers will implement some of the possible exploits described above.

The argument for using NOZZLE, despite the fact that hackers will find ways to confound it, is the same reason that virus scanners are installed almost ubiquitously on computer systems today: it will detect and prevent many known attacks, and as new forms of attacks develop, there are ways to improve its defenses as well. Ultimately, NOZZLE, just like existing virus detectors, is just one layer of a defense in depth.

7 Related Work

This section discusses exploit detection and memory attack prevention.

7.1 Exploit Detection

As discussed, a code injection exploit consists of at least two parts: the NOP sled and shellcode. Detection techniques target either or both of these parts. Signature-based techniques, such as Snort [33], use pattern matching, including possibly regular expressions, to identify attacks that match known attacks in their database. A disadvantage of this approach is that it will fail to detect attacks that are not already in the database. Furthermore, polymorphic malware potentially vary its shellcode on every infection attempt, reducing the effectiveness of pattern-based techniques. Statistical techniques, such as Polygraph [27], address this problem by using improbable properties of the shellcode to identify an attack.

The work most closely related to NOZZLE is Abstract Payload Execution (APE), by Toth and Kruegel [43], and STRIDE, by Akritidis et al. [4, 30], both of which present methods for NOP sled detection in network traffic. APE examines sequences of bytes using a technique they call *abstract execution* where the bytes are considered valid

and correct if they represent processor instructions with legal memory operands. APE identifies sleds by computing the execution length of valid and correct sequences, distinguishing attacks by identifying sufficiently long sequences.

The authors of STRIDE observe that by employing jumps, NOP sleds can be effective even with relatively short valid and correct sequences. To address this problem, they consider all possible subsequences of length n , and detect an attack only when all such subsequences are considered valid. They demonstrate that STRIDE handles attacks that APE does not, showing also that tested over a large corpus of URIs, their method has an extremely low false positive rate. One weakness of this approach, acknowledged by the authors is that "...a worm writer could blind STRIDE by adding invalid instruction sequences at suitable locations..." ([30], p. 105).

NOZZLE also identifies NOP sleds, but it does so in ways that go beyond previous work. First, prior work has not considered the specific problem of sled detection in heap objects or the general problem of heap spraying, which we do. Our results show that applying previous techniques to heap object results in high false positive rates. Second, because we target heap spraying specifically, our technique leverages properties of the entire heap and not just individual objects. Finally, we introduce the concept of surface area as a method for prioritizing the potential threat of an object, thus addressing the STRIDE weakness mentioned above.

7.2 Memory Attack Prevention

While NOZZLE focuses on detecting heap spraying based on object and heap properties, other techniques take different approaches. Recall that heap spraying requires an additional memory corruption exploit, and one method of preventing a heap-spraying attack is to ignore the spray altogether and prevent or detect the initial corruption error. Techniques such as control flow integrity [1], write integrity testing [3], data flow integrity [9], and program shepherding [17] take this approach. Detecting all such possible exploits is difficult and, while these techniques are promising, their overhead has currently prevented their widespread use.

Some existing operating systems also support mechanisms, such as Data Execution Prevention (DEP) [21], which prevent a process from executing code on specific pages in its address space. Implemented in either software or hardware (via the no-execute or "NX" bit), execution protection can be applied to vulnerable parts of an address space, including the stack and heap. With DEP turned on, code injections in the heap cannot execute.

While DEP will prevent many attacks, we believe that NOZZLE is complementary to DEP for the following rea-

sons. First, security benefits from defense-in-depth. For example, attacks that first turn off DEP have been published, thereby circumventing its protection [37]. Second, compatibility issues can prevent DEP from being used. Despite the presence of NX hardware and DEP in modern operating systems, existing commercial browsers, such as Internet Explorer 7, still ship with DEP disabled by default [13]. Third, runtime systems that perform just-in-time (JIT) compilation may allocate JITed code in the heap, requiring the heap to be executable. Finally, code injection spraying attacks have recently been reported in areas other than the heap where DEP cannot be used. Sotirov and Dowd describe spraying attacks that introduce exploit code into a process address space via embedded .NET User Controls [42]. The attack, which is disguised as one or more .NET managed code fragments, is loaded in the process text segment, preventing the use of DEP. In future work, we intend to show that NOZZLE can be effective in detecting such attacks as well.

8 Conclusions

We have presented NOZZLE, a runtime system for detecting and preventing heap-spraying security attacks. Heap spraying has the property that the actions taken by the attacker in the spraying part of the attack are legal and type safe, allowing such attacks to be initiated in JavaScript, Java, or C#. Attacks using heap spraying are on the rise because security mitigations have reduced the effectiveness of previous stack and heap-based approaches.

NOZZLE is the first system specifically targeted at detecting and preventing heap-spraying attacks. NOZZLE uses lightweight runtime interpretation to identify specific suspicious objects in the heap and maintains a global heap health metric to achieve low false positive and false negative rates, as measured using 12 published heap spraying attacks, 2,000 synthetic malicious exploits, and 150 highly-visited benign web sites. We show that with sampling, the performance overhead of NOZZLE can be reduced to 7%, while maintaining low false positive and false negative rates. Similar overheads are observed when NOZZLE is applied to Adobe Acrobat Reader, a recent target of heap spraying attacks. The fact that NOZZLE was able to thwart a real published exploit when applied to the Adobe Reader binary, without requiring any modifications to our instrumentation techniques, demonstrates the generality of our approach.

While we have focused our experimental evaluation on heap-spraying attacks exclusively, we believe that our techniques are more general. In particular, in future work, we intend to investigate using our approach to detect a variety of exploits that use code masquerading as data, such as images, compiled bytecode, etc. [42].

In the future, we intend to further improve the selectivity of the NOZZLE local detector, demonstrate NOZZLE's effectiveness for attacks beyond heap spraying, and further tune NOZZLE's performance. Because heap-spraying attacks can be initiated in type-safe languages, we would like to evaluate the cost and effectiveness of incorporating NOZZLE in a garbage-collected runtime. We are also interested in extending NOZZLE from detecting heap-spraying attacks to tolerating them as well.

Acknowledgements

We thank Emery Berger, Martin Bertscher, Silviu Calinoiu, Trishul Chilimbi, Tal Garfinkel, Ted Hart, Karthik Pattabiraman, Patrick Stratton, and Berend-Jan "SkyLined" Wever for their valuable feedback during the development of this work. We also thank our anonymous reviewers for their comments.

References

- [1] M. Abadi, M. Budiu, Úlfar Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the Conference on Computer and Communications Security*, pages 340–353, 2005.
- [2] A. V. Aho, M. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2007.
- [3] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 263–277, 2008.
- [4] P. Akritidis, E. P. Markatos, M. Polychronakis, and K. G. Anagnostakis. STRIDE: Polymorphic sled detection through instruction sequence analysis. In R. Sasaki, S. Qing, E. Okamoto, and H. Yoshiura, editors, *Proceedings of Security and Privacy in the Age of Ubiquitous Computing*, pages 375–392. Springer, 2005.
- [5] Alexa Inc. Global top sites. http://www.alexa.com/site/ds/top_sites, 2008.
- [6] Arr1val. Exploit made by Arr1val proved in Adobe 9.1 and 8.1.4 on Linux. <http://downloads.securityfocus.com/vulnerabilities/exploits/34736.txt>, Feb. 2009.
- [7] E. D. Berger and B. G. Zorn. DieHard: probabilistic memory safety for unsafe languages. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 158–168, 2006.
- [8] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium*, pages 105–120. USENIX, Aug. 2003.

- [9] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, pages 147–160, 2006.
- [10] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: attacks and defenses for the vulnerability of the decade. *Foundations of Intrusion Tolerant Systems*, pages 227–237, 2003.
- [11] ECMA. ECMAScript language specification. <http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf>, 1999.
- [12] J. C. Foster. *Metasploit Toolkit for Penetration Testing, Exploit Development, and Vulnerability Research*. Syngress Publishing, 2007.
- [13] M. Howard. Update on Internet Explorer 7, DEP, and Adobe software. http://blogs.msdn.com/michael_howard/archive/2006/12/12/update-on-internet-explorer-7-dep-and-adobe-software.aspx, 2006.
- [14] G. Hunt and D. Brubacher. Detours: Binary interception of Win32 functions. In *Proceedings of the USENIX Windows NT Symposium*, pages 135–143, 1999.
- [15] iDefense Labs. Annual vulnerability challenge. <http://labs.iddefense.com/vcp/challenge.php>, 2007.
- [16] I.-K. Kim, K. Kang, Y. Choi, D. Kim, J. Oh, and K. Han. A practical approach for detecting executable codes in network traffic. In S. Ata and C. S. Hong, editors, *Proceedings of Managing Next Generation Networks and Services*, volume 4773 of *Lecture Notes in Computer Science*, pages 354–363. Springer, 2007.
- [17] V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the USENIX Security Symposium*, pages 191–206, 2002.
- [18] J. Leyden. Phatbot arrest throws open trade in zombie PCs. http://www.theregister.co.uk/2004/05/12/phatbot_zombie_trade, May 2004.
- [19] B. Livshits and W. Cui. Spectator: Detection and containment of JavaScript worms. In *Proceedings of the USENIX Annual Technical Conference*, pages 335–348, June 2008.
- [20] A. Marinescu. Windows Vista heap management enhancements. In *BlackHat US*, 2006.
- [21] Microsoft Corporation. Data execution prevention. <http://technet.microsoft.com/en-us/library/cc738483.aspx>, 2003.
- [22] Microsoft Corporation. Microsoft Security Bulletin MS07-017. <http://www.microsoft.com/technet/security/Bulletin/MS07-017.msp>, Apr. 2007.
- [23] Microsoft Corporation. Microsoft Security Advisory (961051). <http://www.microsoft.com/technet/security/advisory/961051.msp>, Dec. 2008.
- [24] Mozilla Developer Center. Crash reporting page. https://developer.mozilla.org/En/Crash_reporting, 2008.
- [25] Mozilla Security Group. Mozilla security bug bounty program. <http://www.mozilla.org/security/bug-bounty.html>, 2004.
- [26] Multi-State Information Sharing and Analysis Center. Vulnerability in Adobe Reader and Adobe Acrobat could allow remote code execution. <http://www.msisac.org/advisories/2009/2009-008.cfm>, Feb. 2009.
- [27] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 226–241, 2005.
- [28] J. D. Pincus and B. Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security and Privacy*, 2(4):20–27, 2004.
- [29] P. P. Pirinen. Barrier techniques for incremental tracing. In *Proceedings of the International Symposium on Memory Management*, pages 20–25, 1998.
- [30] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Emulation-based detection of non-self-contained polymorphic shellcode. In *Proceedings of Symposium on Recent Advances in Intrusion Detection*, pages 87–106, 2007.
- [31] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Network-level polymorphic shellcode detection using emulation. *Journal in Computer Virology*, 2(4):257–274, 2007.
- [32] P. Ratanaworabhan, B. Livshits, and B. Zorn. Nozzle: A defense against heap-spraying code injection attacks. Technical Report MSR-TR-2008-176, Microsoft Research, Nov. 2008.
- [33] M. Roesch. Snort - lightweight intrusion detection for networks. In *Proceedings of the USENIX conference on System administration*, pages 229–238, 1999.
- [34] Samy. The Samy worm. <http://namb.la/popular/>, Oct. 2005.
- [35] B. Schwarz, S. Debray, and G. Andrews. Disassembly of executable code revisited. *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*, pages 45–54, 2002.
- [36] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the Conference on Computer and Communications Security*, pages 298–307, 2004.
- [37] Skape and Skywing. Bypassing windows hardware-enforced DEP. *Uninformed Journal*, 2(4), Sept. 2005.
- [38] SkyLined. Internet Explorer IFRAME src&name parameter BoF remote compromise. http://skypher.com/wiki/index.php?title=Www.edup.tudelft.nl/~bjwever/advisory/_iframe.html.php, 2004.
- [39] SkyLined. Personal communication, 2009.
- [40] Sophos Inc. Stopping zombies, botnets and other email- and web-borne threats. <http://blogs.piercelaw.edu/tradesecretsblog/SophosZombies072507.pdf>, 12 2006.
- [41] A. Sotirov. Heap feng shui in JavaScript. In *Proceedings of Blackhat Europe*, 2007.
- [42] A. Sotirov and M. Dowd. Bypassing browser memory pro-

- tections. In *Proceedings of BlackHat*, 2008.
- [43] T. Toth and C. Krügel. Accurate buffer overflow detection via abstract payload execution. In *Proceedings of Symposium on Recent Advances in Intrusion Detection*, pages 274–291, 2002.
- [44] R. van den Heetkamp. Heap spraying. <http://www.0x000000.com/index.php?i=412&bin=110011100>, Aug. 2007.
- [45] Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. King. Automated web patrol with Strider HoneyMonkeys: Finding web sites that exploit browser vulnerabilities. In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS 2006)*, Feb. 2006.

Cross-Origin JavaScript Capability Leaks: Detection, Exploitation, and Defense

Adam Barth
UC Berkeley
abarth@eecs.berkeley.edu

Joel Weinberger
UC Berkeley
jww@cs.berkeley.edu

Dawn Song
UC Berkeley
dawnsong@cs.berkeley.edu

Abstract

We identify a class of Web browser implementation vulnerabilities, *cross-origin JavaScript capability leaks*, which occur when the browser leaks a JavaScript pointer from one security origin to another. We devise an algorithm for detecting these vulnerabilities by monitoring the “points-to” relation of the JavaScript heap. Our algorithm finds a number of new vulnerabilities in the open-source WebKit browser engine used by Safari. We propose an approach to mitigate this class of vulnerabilities by adding access control checks to browser JavaScript engines. These access control checks are backwards-compatible because they do not alter semantics of the Web platform. Through an application of the inline cache, we implement these checks with an overhead of 1–2% on industry-standard benchmarks.

1 Introduction

In this paper, we identify a class of Web browser implementation vulnerabilities, which we refer to as *cross-origin JavaScript capabilities leaks*, and develop systematic techniques for detecting, exploiting, and defending against these vulnerabilities. An attacker who exploits a cross-origin JavaScript capability leak can inject a malicious script into an honest Web site’s security origin. These attacks are more severe than cross-site scripting (XSS) attacks because they affect all Web sites, including those free of XSS vulnerabilities. Once an attacker can run script in an arbitrary security origin, the attacker can, for example, issue transactions on the user’s bank account, regardless of any SSL encryption, cross-site scripting filter, or Web application firewall.

We observe that these cross-origin JavaScript capability leaks are caused by an architectural flaw shared by most modern Web browsers: the Document Object Model (DOM) and the JavaScript engine enforce the same-origin policy using two different security models. The DOM uses an access control model, whereas the JavaScript engine uses object-capabilities.

- **Access Control.** The DOM enforces the same-origin policy using a reference monitor that prevents one Web site from accessing resources allocated to another Web site. For example, whenever

a script attempts to access the cookie database, the DOM checks whether the script’s security origin has sufficient privileges to access the cookies.

- **Object-Capabilities.** The JavaScript engine enforces the same-origin policy using an object-capability discipline that prevents one Web site from obtaining JavaScript pointers to sensitive objects that belong to a foreign security origin. Without JavaScript pointers to sensitive objects in foreign security origins, malicious scripts are unable to interfere with those objects.

Most modern Web browsers, including Internet Explorer, Firefox, Safari, Google Chrome, and Opera, use this design. However, the design’s mismatch in enforcement paradigms leads to vulnerabilities whenever the browser leaks a JavaScript pointer from one security origin to another. Once a malicious script gets a JavaScript pointer to an honest JavaScript object, the attacker can leverage the object-capability security model of the JavaScript engine to escalate its DOM privileges. With escalated DOM privileges, the attacker can completely compromise the honest security origin by injecting a malicious script into the honest security origin.

To study this class of vulnerabilities, we devise an algorithm for detecting individual cross-origin JavaScript capability leaks. Using this algorithm, we uncover new instances of cross-origin JavaScript capability leaks in the WebKit browser engine used by Safari. We then illustrate how an attack can abuse these leaked JavaScript pointers by constructing proof-of-concept exploits. We propose defending against cross-origin JavaScript capability leaks by harmonizing the security models used by the DOM and the JavaScript engine.

- **Leak Detection.** We design an algorithm for automatically detecting cross-origin JavaScript capability leaks by monitoring the “points-to” relation among JavaScript objects in the heap. From this relation, we define the security origin of each JavaScript object by tracing its “prototype chain.” We then search the graph for edges that connect objects in one security origin with objects in another security origin. These *suspicious edges* likely represent cross-origin JavaScript capability leaks.

- **Vulnerabilities and Exploitation.** We implement our leak detection algorithm and find two new high-severity cross-origin JavaScript capability leaks in WebKit. Although these vulnerabilities are implementation errors in WebKit, the presence of the bugs illustrates the fragility of the general architecture. (Other browsers have historically had similar vulnerabilities [17, 18, 19].) We detail these vulnerabilities and construct proof-of-concept exploits to demonstrate how an attacker can leverage a leaked JavaScript pointer to inject a malicious script into an honest security origin.
- **Defense.** We propose that browser vendors proactively defend against cross-origin JavaScript capability leaks by implementing access control checks throughout the JavaScript engine instead of reactively plugging each leak. Adding access control checks to the JavaScript engine addresses the root cause of these vulnerabilities (the mismatch between the security models used by the DOM and by the JavaScript engine) and provides defense-in-depth in the sense that both an object-capability and an access control failure are required to create an exploitable vulnerability. This defense is perfectly backwards-compatible because these access checks do not alter the semantics of the Web platform. Our implementation of these access control checks in WebKit incurs an overhead of only 1–2% on industry-standard benchmarks.

Contributions. We make the following contributions:

- We identify a class of Web browser implementation vulnerabilities: cross-origin JavaScript capability leaks. These vulnerabilities arise when the browser leaks a JavaScript pointer from one security origin to another security origin.
- We introduce an algorithm for detecting cross-origin JavaScript capability leaks by monitoring the “points-to” relation of the JavaScript heap. Our algorithm uses a graph-based definition of the security origin of a JavaScript object.
- We reveal cross-origin JavaScript capability leaks and demonstrate techniques for exploiting these vulnerabilities. These exploits rely on the mismatch between the DOM’s access control security model and the JavaScript engine’s object-capability security model.
- We propose that browsers defend against cross-origin JavaScript capability leaks by implementing access control checks in the JavaScript engine. This defense is perfectly backwards-compatible and achieves a low overhead of 1–2%.

Organization. This paper is organized as follows. Section 2 identifies cross-origin JavaScript capability

leaks as a class of vulnerabilities. Section 3 presents our algorithm for detecting cross-origin JavaScript capability leaks. Section 4 details the individual vulnerabilities we uncover with our algorithm and outlines techniques for exploiting these vulnerabilities. Section 5 proposes defending against cross-origin JavaScript capability leaks by adding access control checks to the JavaScript engine. Section 6 relates our work to the literature. Section 7 concludes.

2 JavaScript Capability Leaks

In this section, we describe our interpretation of JavaScript pointers as object-capabilities and identify cross-origin JavaScript capability leaks as a class of implementation vulnerabilities in browsers. We then sketch how these vulnerabilities are exploited and the consequences of a successful exploit.

2.1 Object-Capabilities

In modern Web browsers, the JavaScript engine enforces the browser’s same-origin policy using an object-capability discipline: a script can obtain pointers only to JavaScript objects created by documents in its security origin. A script can obtain JavaScript pointers to JavaScript objects either by accessing properties of JavaScript object to which the script already has a JavaScript pointer or by conjuring certain built-in objects such as the global object and `Object.prototype` [14]. As in other object-capability systems, the ability to influence an object is tied to the ability to designate the object. In browsers, a script can manipulate a JavaScript object only if the script has a pointer to the object. Without a pointer to an object in a foreign security origin, a malicious script cannot influence honest JavaScript objects and cannot interfere with honest security origins.

One exception to this object-capability discipline is the JavaScript global object. According to the HTML 5 specification [10], the global object (also known as the window object) is visible to foreign security origins. There are a number of APIs for obtaining pointers to global objects from foreign security origins. For example, the `contentWindow` property of an `<iframe>` element is the global object of the document contained in the frame. Unlike most JavaScript objects, the global object is also a DOM object (called `window`) and is equipped with a reference monitor that prevents scripts in foreign security origins from getting or setting arbitrary properties of the object. This reference monitor does not forbid all accesses because some are desirable. For example, the `postMessage` method [10] is exposed across origins to facilitate mashups [1]. These exposed properties complicate the enforcement of the same-origin policy, which can lead to vulnerabilities.

2.2 Capability Leaks

Browsers occasionally contain bugs that leak JavaScript pointers from one security origin to another. These vulnerabilities are easy for developers to introduce into browsers because the DOM contains pointers to JavaScript objects in multiple security origins and developers can easily select the wrong pointer to disclose to a script. We identify these vulnerabilities as a class, which we call *cross-origin JavaScript capabilities leaks*, because they follow a common pattern. Identifying this class lets us analyze the concepts common to these vulnerabilities in all browsers.

The JavaScript language makes pointer leaks particularly devastating for security because JavaScript objects inherit many of their properties from a *prototype* object. When a script accesses a property of an object, the JavaScript engine uses the following algorithm to look up the property:

- If the object has the property, return its value.
- Otherwise, look up the property on the object's prototype (designated by the current object's `__proto__` property).

These prototype objects, in turn, inherit many of their properties from their prototypes in a chain that leads back to the `Object.prototype` object, whose `__proto__` property is `null`. All the objects associated with a given document have a prototype chain that leads back to that document's `Object.prototype` object. Given a JavaScript pointer to an object, a script can traverse this prototype chain by accessing the object's `__proto__` property. In particular, if an attacker obtains a pointer to an honest object, the attacker can obtain a pointer to the honest document's `Object.prototype` object and can influence the behavior of all the other JavaScript objects associated with the honest document.

2.3 Laundries

Once the attacker has obtained a pointer to the `Object.prototype` of an honest document, the attacker has several avenues for compromising the honest security origin. One approach is to abuse powerful functions reachable from `Object.prototype`, which we refer to as *laundries* because they let the attacker “wash away” his or her agency (analogous to laundering money). These functions often call one or more DOM APIs, letting the attacker call these APIs indirectly. Because these functions are defined by the honest document, the DOM's reference monitor allows the access [10]. However, if the attacker calls these functions with unexpected arguments, the functions might become confused deputies [9] and inadvertently perform the attacker's misdeeds.

Most Web sites contains innumerable laundries. We illustrate how an attacker can abuse a laundry by examining a representative laundry from the Prototype JavaScript library [22]: `invoke`. The `invoke` method is used to call a method, specified by name, on each object contained in an array. The attacker can use this function to trick the honest page into calling a universal DOM method, such as `setTimeout`. Suppose the attacker has a JavaScript pointer to an array named `honest_array` from an honest document that uses the Prototype library (for how this might occur, see Section 4.3) and that `honest_window` is the honest document's global object. The attacker can inject a malicious script into the honest security origin as follows:

```
honest_array.push(honest_window);
honest_array.invoke("setTimeout",
    "... malicious script ...", 0);
```

The attacker first adds the `honest_window` object to the array and then asks the honest principal to call the `setTimeout` method of the `honest_window`. When the JavaScript engine attempts to call the `setTimeout` DOM API, the DOM permits the call because the honest `invoke` method (acting as a confused deputy) issued the call. The DOM then runs the malicious script supplied by the attacker in the honest security origin.

2.4 Consequences

Once the attacker is able to run a malicious script in the honest security origin, all the browser's cross-origin security protections evaporate. The situation is as if every Web site contained a cross-site scripting vulnerability: the attacker can steal the user's authentication cookie or password, learn confidential information present on the Web site (e.g., read email messages on a webmail site), and issue transactions on behalf of the user (e.g., transfer money out of the user's bank account). Because these cross-origin JavaScript capability leaks are browser vulnerabilities, there is little a Web site can do to defend itself against these attacks.

3 JavaScript Capability Leak Detection

In this section, we describe the design and implementation of an algorithm for detecting cross-origin JavaScript capability leaks. Although the algorithm has a modest overhead, our instrumented browser performs comparably to Safari 3.1, letting us analyze complex Web applications.

3.1 Design

Assigning Security Origins. To detect cross-origin JavaScript capability leaks, we monitor the *heap graph*, the “points-to” relation between JavaScript objects in the

JavaScript heap (see Section 3.2 for details about the “points-to” relation). We annotate each JavaScript object in the heap graph with a security origin indicating which security origin “owns” the object. We compute the security origin of each object directly from the “is-prototype-of” relation in the heap graph using the following algorithm:

1. Let `obj` be the JavaScript object in question.
2. If `obj` was created with a non-null prototype, assign `obj` the same origin as its prototype.
3. Otherwise, `obj` must be the object prototype for some document d . In that case, assign `obj` the security origin of d (i.e., the scheme, host, and port of that d ’s URL).

This algorithm is unambiguous because, when created, each JavaScript object has a unique prototype, identified by its `__proto__` property. Although an object’s `__proto__` can change over time, we fix the security origin of an object at creation-time.

Minimal Capabilities. This algorithm for assigning security origins to objects is well-suited to analyzing leaks of JavaScript pointers for two reasons. First, the algorithm is defined largely without reference to the DOM, letting us catch bugs in the DOM. Second, the algorithm reflects an object-capability perspective in that each JavaScript object is a strictly more powerful object-capability than the `Object.prototype` object that terminates its prototype chain. An attacker with a JavaScript pointer to the object can follow the object’s prototype chain by repeatedly dereferencing the object’s `__proto__` property and eventually obtain a JavaScript pointer to the `Object.prototype` object. In these terms, we view the `Object.prototype` object as the “minimal object-capability” of an origin.

Suspicious Edges. After annotating the heap graph with the security origin of each object, we detect a leaked JavaScript pointer as an edge from an object in one security origin to an object in another security origin. These *suspicious edges* represent failures of the JavaScript engine to segregate JavaScript objects into distinct security origins. Not all of these suspicious edges are actually security vulnerabilities because the HTML specification requires some JavaScript objects, such as the global object, be visible to foreign security origins. To prevent exploits, browsers equip these objects with a reference monitor that prevents foreign security origins from getting or setting arbitrary properties of the object. In addition to the global object, a handful of other JavaScript objects required to be visible to foreign security origins. These objects are annotated in WebKit’s Interface Description Language (IDL) with the attribute `DoNotCheckDomainSecurity`.

3.2 The “Points-To” Relation

In our heap graph, we include two kinds of points in the “points-to” relation: explicit pointers that are stored as properties of JavaScript objects and implicit pointers that are stored internally by the JavaScript engine.

Explicit Pointers. A script can alter the properties of an object using the `get`, `set`, and `delete` operations.

- `get` looks up the value of an object property.
- `set` alters the value of an object property.
- `delete` removes a property from an object.

To monitor the “points-to” relation between JavaScript objects in the JavaScript heap, we instrument the `set` operation. Whenever the JavaScript engine invokes the `set` operation to store a JavaScript object in a property of another JavaScript object, we add an edge between the two objects in our representation of the heap graph. If the `set` operation overwrites an existing property, we remove the obsolete edge from the graph. To improve performance, we ignore JavaScript values because JavaScript values cannot hold JavaScript pointers and therefore are leaves in the heap graph. We remove JavaScript objects from the heap graph when the objects are deleted by the JavaScript garbage collector.

Implicit Pointers. The above instrumentation does not give us a complete picture of the “points-to” relation in the JavaScript heap because the operational semantics of the JavaScript language [14] rely on a number of implicit JavaScript pointers, which are not represented explicitly as properties of a JavaScript object. For example, consider the following script:

```
var x = ...
function f() {
  var y = ...
  function g() {
    var z = ...
    function h() { ... }
  }
}
```

Function `h` can obtain the JavaScript pointers stored in variables `x`, `y`, and `z` even though there are no JavaScript pointers between `h` and these objects. The function `h` can obtain these JavaScript pointers because the algorithm for resolving variable names makes use of an implicit “next” pointer that connects `h`’s scope object to the scope objects of `g`, `f`, and the global scope. Instead of being stored as properties of JavaScript objects, these implicit pointers are stored as member variables of native objects in the JavaScript engine. To improve the completeness of our heap graph, we include these implicit JavaScript pointers explicitly as edges between the JavaScript scope objects.

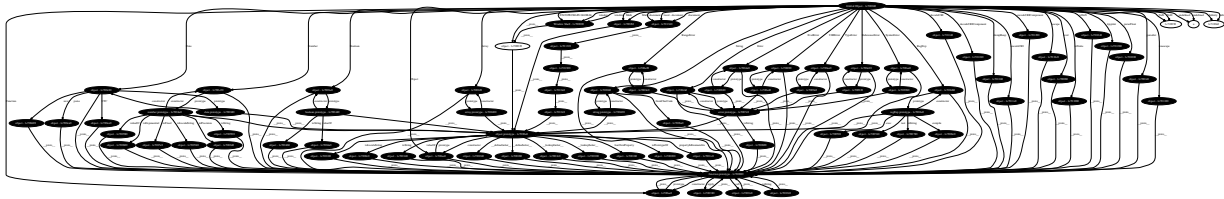


Figure 1: The heap graph of an empty document.

3.3 Implementation

We implemented our leak detection algorithm in a 1,393 line patch to WebKit's Nitro JavaScript engine. Our algorithm can construct heap graphs of complex Web applications, such as Gmail or the Apple Store. For example, one heap graph of a Gmail inbox contains 54,140 nodes and 130,995 edges. These graphs are often visually complex and difficult to interpret manually. Figure 1 illustrates the nature of these graphs by depicting the heap graph of an empty document. Although our instrumentation slows down the browser, the instrumented browser is still faster than Safari 3.1, demonstrating that our algorithm scales to complex Web applications.

4 Vulnerabilities and Exploitation

In this section, we use our leak detector to detect cross-origin JavaScript capability leaks in WebKit. After discovering two new vulnerabilities, we illuminate the vulnerabilities by constructing proof-of-concept exploits using three different techniques. In addition, we apply our understanding of JavaScript pointers to breaking the Subspace [11] mashup design.

4.1 Test Suite

To find example cross-origin JavaScript capability leaks, we run our instrumented browser through a test suite. Ideally, to reduce the number of false negatives, we would use a test suite with high coverage. Because our goal is to find example vulnerabilities, we use the WebKit project's regression test suite. This test suite exercises a variety of browser security features and tests for the non-existence of past security vulnerabilities. Using this test suite, our instrumentation found two new high-severity cross-origin JavaScript capability leaks. Instead of attempting to discover and patch all these leaks, we recommend a more comprehensive defense, detailed in Section 5.

WebKit's regression test suite uses a JavaScript object named `layoutTestController` to facilitate its tests. For example, each tests notifies the testing harness that the test is complete by calling the `notifyDone` method of the `layoutTestController`. We modified this `notifyDone` method to store the JavaScript heap graph in the file system after each test completes.

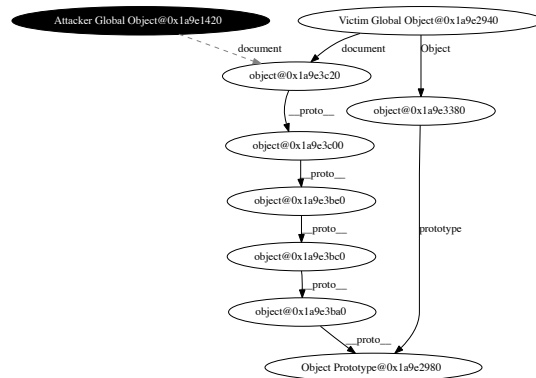


Figure 2: Selected nodes from a heap graph showing a cross-origin JavaScript capability leak of the document object, `object@0x1a9e3c20`, after a navigation.

The `layoutTestController` contains a number of objects that are shared between all security origins. Our instrumentation flags JavaScript pointers to these objects as suspicious, and, in fact, these pointers are exploitable in the test configuration of the browser. However, these pointers are not present in the release configuration of the browser because the `layoutTestController` itself is present only during testing. We white listed these objects as visible to multiple security origins.

4.2 Navigation and Document

Vulnerability. When the browser navigates a window from one Web page to another, the browser replaces the document originally displayed in the window with a new document retrieved from the network. Our instrumentation found that WebKit leaks a JavaScript pointer to the new document object every time a window navigates because the DOM updates the `document` property of the old global object to point to the new document occupying the frame. This leak is visible in the heap graph (see Figure 2) as a dashed line from Attacker Global Object@0x1a9e1420 to the honest document object, `object@0x1a9e3c20`.

Exploit. Crafting an exploit for this vulnerability is subtle. An attacker cannot simply hold a JavaScript pointer to the old global object and access its `document` property because all JavaScript pointers to global objects are updated to the new global object when a frame is navigated navigation [10]. However, the properties of the old global object are still visible to functions defined by the old document via the scope chain as global variables. In particular, an attacker can exploit this vulnerability as follows:

1. Create an `<iframe>` to `http://attacker.com/iframe.html`, which defines the following function in a malicious document:

```
function exploit() {
  var elmt = document.
    createElement("script");
  elmt.src =
    "http://attacker.com/atk.js";
  document.body.appendChild(elmt);
}
```

Notice that the `exploit` function refers to the document as a global variable, `document`, and not as a property of the global object, `window.document`.
2. In the parent frame, store a pointer to the `exploit` function by running the following JavaScript:

```
window.f = frames[0].exploit;
```
3. Navigate the frame to `http://example.com/`.
4. Call the function: `window.f()`.

After the attacker navigates the child frame to `http://example.com/`, the DOM changes the `document` variable in the function `exploit` to point to the honest document object instead of the attacker's document object. The `exploit` function can inject arbitrary script into the honest document using a number of standard DOM APIs. Once the attacker has injected script into the honest document, the attacker can impersonate the honest security origin to the browser.

4.3 Lazy Location and History

Vulnerability. For performance, WebKit instantiates the `window.location` and `window.history` objects lazily the first time they are accessed. When instantiating these objects, the browser constructs their prototype chains. In some situations, WebKit constructs an incorrect prototype chain that connects these objects to the `Object.prototype` of a foreign security origin, creating a vulnerability if, for example, a document uses the following script to “frame bust” [12] in order to avoid clickjacking [7] attacks:

```
top.location.href =
  "http://example.com/";
```

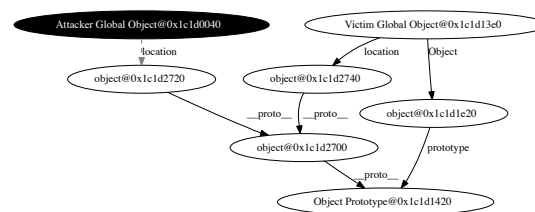


Figure 3: Selected nodes from a heap graph showing a cross-origin JavaScript capability leak of the location prototype, `object@0x1c1d2700`, to the attacker after the victim attempts to frame bust.

This line of JavaScript changes the location of the top-most frame, navigating that frame to a trusted Web site. The browser permits cross-origin access to a frame's location object to allow navigation [1]. If this script is the first script to access the location object of the top frame, then WebKit will mistakenly connect the top frame's newly constructed location object to the `Object.prototype` of the child frame (instead of to the `Object.prototype` of the top frame) because the child frame is currently in scope lexically.

Exploit. To exploit this cross-origin JavaScript capability leak, the attacker proceeds in two phases: (1) the attacker obtains a JavaScript pointer to the honest `Object.prototype`, and (2) the attacker abuses the honest `Object.prototype` to inject a malicious script into the honest security origin. To obtain a JavaScript pointer to the honest `Object.prototype`, the attacker create an `<iframe>` to an honest document that frame busts and runs the following script in response to the `beforeunload` event:

```
var location_prototype =
  window.location.__proto__;
var honest_object_prototype =
  location_prototype.__proto__;
```

Because the `beforeunload` event handler runs after the child frame has attempted to frame bust, the attacker's location object has been instantiated by the honest document and is mistakenly attached to the honest `Object.prototype` (see Figure 3). The attacker obtains a pointer to the honest `Object.prototype` by traversing this prototype chain.

Once the attacker has obtained a JavaScript pointer to the honest `Object.prototype`, there are a number of techniques the attacker can use to compromise the honest security origin completely. We describe two representative examples:

1. Many Web sites use JavaScript libraries to smooth over incompatibilities between browsers and reuse

common code. One of the more popular JavaScript libraries is the Prototype library [22], which is used by CNN, Apple, Yelp, Digg, Twitter, and many others. If the honest page uses the Prototype library, the attacker can inject arbitrary script into the honest page by abusing the powerful `invoke` function defined by the Prototype library. For example, the attacker can use the follow script:

```
var honest_function =
    honest_object_prototype.
    __defineGetter__;
var honest_array =
    honest_function.
    argumentNames();
honest_array.push(frames[0]);
honest_array.invoke("setTimeout",
    "... malicious script ...");
```

In the Prototype library, arrays contain a method named `invoke` that calls the method named by its first argument on each element of its array, passing the remaining arguments to the invoked method. To abuse this method, the attacker first obtains a pointer to an honest array object by calling the `argumentNames` method of an honest function reachable from the `honest_object_prototype` object. The attacker then pushes the global object of the child frame onto the array and calls the honest document's `setTimeout` method via `invoke`. The honest global object has a reference monitor that prevents the attacker from accessing `setTimeout` directly, but the reference monitor allows `invoke` to access `setTimeout` because `invoke` is defined by the honest document.

2. Even if the honest Web page does not use a complex JavaScript library, the attacker can often find a snippet of honest script to trick. For example, suppose the attacker installs a “setter” function for the `foo` property of the honest `Object.prototype` as follows:

```
function evil(x) {
    x.innerHTML =
        '';
};
honest_object_prototype.
    __defineSetter__('foo', evil);
```

Now, if the honest script stores a DOM node in a property of an object as follows:

```
var obj = new Object();
obj.foo = honest_dom_node;
```

The JavaScript engine will call the attacker's setter function instead of storing `honest_dom_node` into the `foo` property of `obj`, causing the variable `x` to contain a JavaScript pointer to `honest_dom_node`. Once the attacker's function is called with a pointer to the honest DOM node, the attacker can inject malicious script into the honest document using the `innerHTML` API.

4.4 Capability Leaks in Subspace

The Subspace mashup design [11] lets a trusted integrator communicate with an untrusted gadget by passing a JavaScript pointer from the integrator to the gadget:

A Subspace JavaScript object is created in the top frame and passed to the mediator frame... The mediator frame still has access to the Subspace object it obtained from the top frame, and passes this object to the untrusted frame. [11]

Unfortunately, the Subspace design relies on leaking a JavaScript pointer from a trusted security origin to an untrusted security origin, creating a cross-origin JavaScript capability leak. By leaking the communication object, Subspace also leaks a pointer to the trusted `Object.prototype` via the prototype chain of the communication object.

To verify this attack, we examined CrossSafe [25], a public implementation of Subspace. We ran a Cross-Safe tutorial in our instrumented browser and examined the resulting heap graph. Our detector found a cross-origin JavaScript capability leak: the `channel` object is leaked from the integrator to the gadget. By repeatedly dereferencing the `__proto__` property, the untrusted gadget can obtain a JavaScript pointer to the trusted `Object.prototype` object. The untrusted gadget can then inject a malicious script into the trusted integrator using one of the techniques described in Section 4.3.

5 Defense

In this section, we propose a principled defense for cross-origin JavaScript capability leaks. Our defense addresses the root cause of these vulnerabilities and incurs a minor performance overhead.

5.1 Approach

Currently, browser vendors defend against cross-origin JavaScript capability leaks by patching each individual leak after the leak is discovered. We recommend another approach for defending against these vulnerabilities: add access control checks throughout the JavaScript engine. We recommend this principled approach over ad-hoc leak plugging for two reasons:

- This approach addresses the core design issue underlying cross-origin JavaScript capability leak vulnerabilities: the mismatch between the DOM's access control security model and the JavaScript engine's object-capability security model.
- This approach provides a second layer of defense: if the browser is leak-free, all the access control checks will be redundant and pass, but if the browser contains a leak, the access control checks prevent the attacker from exploiting the leak.

In a correctly implemented browser, Web content will be unable to determine whether the browser implements the access control checks we recommend. The additional access control checks enhance the mechanism used to enforce the same-origin policy but do not alter the policy itself, resulting in zero compatibility impact.

Another plausible approach to mitigating these vulnerabilities is to adopt an object-capability discipline throughout the DOM. This approach mitigates the severity of cross-origin JavaScript capability leaks by limiting the damage an attacker can wreak with the leaked capability. For example, if the browser leaks an honest history object to the attacker, the attacker would be able to manipulate the history object, but would not be able to alter the document object. Conceptually, either adding access control checks to the JavaScript engine or adopting an object-capability discipline throughout the DOM resolves the underlying architectural security issue, but we recommend adopting the access control paradigm for two main reasons:

- Adopting an object-capability discipline throughout the DOM requires “taming” [15] the DOM API. The current DOM API imbues every DOM node with the full authority of the node's security origin because the API exposes a number of “universal” methods, such as `innerHTML` that can be used to run arbitrary script. Other researchers have designed capability-based DOM APIs [4], but taming the DOM API requires a number of non-backwards compatible changes. A browser that makes these changes will be unpopular because the browser will be unable to display a large fraction of Web sites.
- The JavaScript language itself has a number of features that make enforcing an object-capability discipline challenging. For example, every JavaScript object has a prototype chain that eventually leads back to the `Object.prototype`, making it difficult to create a weaker object-capability than the `Object.prototype`. Unfortunately, the `Object.prototype` itself represents a powerful object-capability with the ability to interfere with the properties of every other object from the same document (e.g., the exploit in Section 4.3.)

Although we recommend that browsers adopt the access control paradigm for Web content, other projects, such as Caja [16] and ADsafe [3], take the opposite approach and elect to enforce an object-capability discipline on the DOM. These projects succeed with this approach because the preceding considerations do not apply: these projects target new code (freeing themselves from backwards compatibility constraints) that is written in a subset of JavaScript (freeing themselves from problematic language features). For further discussion, see Section 6.

5.2 Design

We propose adding access control checks to the JavaScript engine by inserting a reference monitor into each JavaScript object. The reference monitor interposes on each `get` and `set` operation (described in Section 3) and performs the following access control check:

1. Let the *active origin* be the origin of the document that defined the currently executing script.
2. Let the *target origin* be the origin that “owns” the JavaScript object being accessed, as computed by the algorithm in Section 3.1.
3. Allow the access if the browser considers the active origin and the target origin to be the same origin (i.e., if their scheme, hosts, and ports match).
4. Otherwise, deny access.

If the access is denied, the JavaScript engine returns the value `undefined` for `get` operations and simply ignores `set` operations. In addition to adding these access control checks, we record the security origin of each JavaScript object when the object is created. Our implementation does not currently insert access control checks for `delete` operations, but these checks could be added at a minor performance cost. Some JavaScript objects, such as the global object, are visible across origins. For these objects, our reference monitor defers to the reference monitors that already protect these objects.

5.3 Inline Cache

The main disadvantage of performing an access control check for every JavaScript property access is the runtime overhead of performing the checks. Sophisticated Web applications access JavaScript properties an enormous number of times per second, and browser vendors have heavily optimized these code paths. However, we observe that the proposed access checks are largely redundant and amenable to optimization because scripts virtually always access objects from the same origin.

Cutting-edge JavaScript engines, including Safari 4's Nitro JavaScript Engine, Google Chrome's V8 JavaScript engine, Firefox 3.5's TraceMonkey JavaScript engine, and Opera 11's Carakan JavaScript engine, optimize JavaScript property accesses using an

inline cache [24]. (Of the major browser vendors, only Microsoft has yet to announce plans to implement this optimization.) These JavaScript engines group together JavaScript objects with the same “structure” (i.e., whose properties are laid out the same order in memory). When a script accesses a property of an object, the engine caches the object’s group and the memory offset of the property inline in the compiled script. The next time that compiled script accesses a property of an object, the inline cache checks whether the current object has the same structure as the original object. If the two objects have the same structure, a *cache hit*, the engine uses the memory offset stored in the cache to access the property. Otherwise, a *cache miss*, the engine accesses the property using the normal algorithm.

Notice that two objects share the same structure only if their prototypes share the same structure. Additionally, the Nitro JavaScript engine initializes each `Object.prototype` with a unique structure identifier, preventing two object from different security origins (as defined by our prototype-based algorithm) from being grouped together as sharing the same structure. (Other JavaScript engines, such as V8, do contain structure groups that span security origins, but this design is not necessary for performance.) Whenever the inline cache has a hit, we observe the following:

- The current object is from the same security origin as the original object that created the cache entry because the two objects share the same structure.
- The script has the same security origin as when the cache entry was created because the cache is inlined into the script and the security origin of the script is fixed at compile time.

Taken together, these properties imply that the current access control check will return the same result as the original check because both of the origins involved in the check are unchanged. Therefore, we need not perform an access control check during a cache hit, greatly reducing the performance overhead of adding access control checks to the JavaScript engine.

5.4 Evaluation

To evaluate performance overhead of our defense, we added access control checks to Safari 4’s Nitro JavaScript engine in a 394 line patch. We verified that our access control checks actually defeat the proof-of-concept exploits we construct in Section 4. To speed up the access control checks, we represented each security origin by a pointer, letting us allow the vast majority of accesses using a simple pointer comparison. In some rare cases, including to deny access, our implementation performs a more involved access check. The majority of performance overhead in our implementation is caused

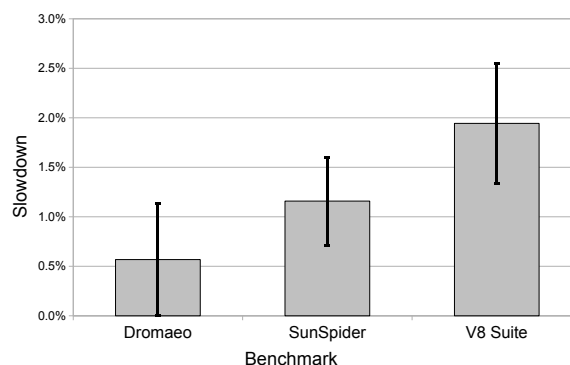


Figure 4: Overhead for access control checks as measure by industry-standard JavaScript benchmarks (average of 10 runs, 95% confidence).

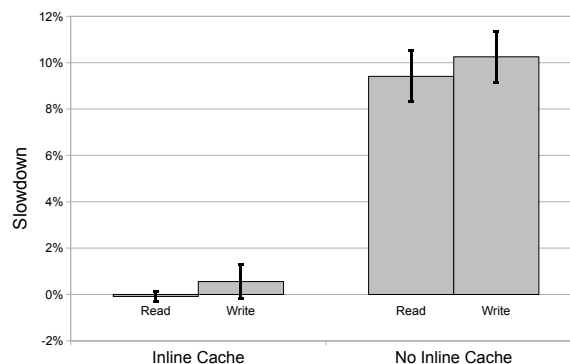


Figure 5: Overhead for reading and writing properties of JavaScript objects both with and without an inline cache as measured by microbenchmarks (average of 10 runs, 95% confidence).

by computing the currently active origin from the lexical scope, which can be reduced with further engineering.

Overall Performance. Our implementation incurs a small overhead on industry-standard JavaScript benchmarks (see Figure 4). On Mozilla’s Dromaeo benchmark, we observed a 0.57% slowdown for access control versus an unmodified browser (average of 10 runs, $\pm 0.58\%$, 95% confidence). On Apple’s SunSpider benchmark, we observed a 1.16% slowdown (average of 10 runs, $\pm 0.45\%$, 95% confidence). On Google’s V8 benchmark, we observed a 1.94% slowdown (average of 10 runs, ± 0.61 , 95% confidence). We hypothesize that the variation in slowdown between these benchmarks is due to the differing balance between arithmetic operations and property accesses in the different benchmarks. Note that these overhead numbers are tiny in comparison with the 338% speedup of Safari 4 over Safari 3.1 [24].

Benefits of Inline Cache. We attribute much of the performance of our access checks to the inline cache, which lets our implementation skip redundant access control checks for repeated property accesses. To evaluate the performance benefits of the inline cache, we created two microbenchmarks, “read” and “write.” In the read benchmark, we repeatedly performed a `get` operation on one property of a JavaScript object in a loop. In the write benchmark, we repeatedly performed a `set` operation on one property of a JavaScript object in a loop. We then measured the slowdown incurred by the access control checks both with the inline cache enabled and with the inline cache disabled (see Figure 5). With the inline cache enabled, we observed a -0.08% slowdown (average of 50 runs, $\pm 0.22\%$, 95% confidence) on the read benchmark and a 0.55% slowdown (average of 50 runs, $\pm 0.74\%$, 95% confidence) on the write benchmark. By contrast, with the inline cache disabled, we observed a 9.41% slowdown (average of 50 runs, $\pm 1.11\%$, 95% confidence) on the read benchmark and a 10.25% slowdown (average of 50 runs, $\pm 1.00\%$, 95% confidence) on the write benchmark.

From these observations we conclude that browser vendors can implement access control checks for every `get` and `set` operation with a performance overhead of less than 1–2%. To reap these security benefits with minimal overhead, the JavaScript engine should employ an inline cache to optimize repeated property accesses, and the inline cache should group structurally similar JavaScript objects only if those objects are from the same security origin.

6 Related Work

The operating system literature has a rich history of work on access control and object-capability systems [13, 21, 23, 8]. In this section, we focus on comparing our work to related work on access control and object-capability systems in Web browsers.

FBJS, Caja, and ADsafe. Facebook, Yahoo!, and Google have developed JavaScript subsets, called FBJS [5], ADsafe [3], and Caja [16], respectively, that enforce an object-capability discipline by removing problematic JavaScript features (such as prototypes) and DOM APIs (such as `innerHTML`). These projects take the opposite approach from this paper: they extend the JavaScript engine’s object-capability security model to the DOM instead of extending the DOM’s access control security model to the JavaScript engine. These projects choose this alternative design point for two reasons: (1) the projects target new social networking gadgets and advertisements that are free from compatibility constraints and (2) these projects are unable to alter legacy browsers because they must work in existing

browsers. We face the opposite constraints: we cannot alter legacy content but we can change the browser. For these reasons, we recommend the opposite design point.

Opus Palladianum. The Opus Palladianum (OP) Web browser [6] isolates security origins into separate sandboxed components. This component-based browser architecture makes it easier to reason about cross-origin JavaScript capability leaks because these capability leaks must occur between browser components instead of within a single JavaScript heap. We can view the sandbox as a coarse-grained reference monitor. Unfortunately, the sandbox alone is too coarse-grained to implement standard browser features such as `postMessage`. To support these features, the OP browser must allow inter-component references, but without a public implementation, we are unable to evaluate whether these inter-component references give rise to cross-origin JavaScript capability leaks.

Script Accenting. Script accenting [2] is a technique for adding defense-in-depth to the browser’s enforcement of the same-origin policy. To mitigate mistaken script execution, the browser encrypts script source code with a key specific to the security origin of the script. Whenever the browser attempts to run a script in a security origin, the browser first decrypts the script with the security origin’s key. If decryption fails, likely because of a vulnerability, the browser refuses to execute the script. Script accenting similarly encrypts the names of JavaScript properties ostensibly preventing a script from manipulating properties of objects from another origin. Unfortunately, this approach is not expressive enough to represent the same-origin policy (e.g., this design does not support `document.domain`). In addition, script accenting requires XOR encryption to achieve sufficient performance, but XOR encryption lacks the integrity protection required to make the scheme secure.

Cross-Origin Wrappers. Firefox 3 uses cross-origin wrappers [20] to mitigate security vulnerabilities caused by cross-origin JavaScript capability leaks. Instead of exposing JavaScript objects directly to foreign security origins, Firefox exposes a “wrapper” object that mediates access to the wrapped object with a reference monitor. Implementing cross-origin wrappers correctly is significantly more complex than implementing access control correctly because the cross-origin wrappers must wrap and unwrap objects at the appropriate times in addition to implementing all the same access control checks. Our access control design can be viewed as a high-performance technique for reducing this complexity (and the attendant bugs) by adding the reference monitor to every object.

7 Conclusions

In this paper, we identify a class of vulnerabilities, *cross-origin JavaScript capability leaks*, that arise when the browser leaks a JavaScript pointer from one security origin to another. These vulnerabilities undermine the same-origin policy and prevent Web sites from securing themselves against Web attackers. We present an algorithm for detecting cross-origin JavaScript capability leaks by monitoring the “points-to” relation between JavaScript objects in the JavaScript heap. We implement our detection algorithm in WebKit and use it to find new cross-origin JavaScript capability leaks by running the WebKit regression test suite in our instrumented browser. Having discovered these leaked pointers, we turn our attention to exploiting these vulnerabilities. We construct exploits to illustrate the vulnerabilities and find that the root cause of the these vulnerabilities is the mismatch in security models between the DOM, which uses access control, and the JavaScript engine, which uses object-capabilities. Instead of patching each leak, we recommend that browser vendors repair the underlying architectural issue by implementing access control checks throughout the JavaScript engine. Although a straight-forward implementation that performed these checks for every access would have a prohibitive overhead, we demonstrate that a JavaScript engine optimization, the inline cache, reduces this overhead to 1–2%.

Acknowledgements. We thank Chris Karloff, Oliver Hunt, Collin Jackson, John C. Mitchell, Rachel Parke-Houben, and Sam Weinig for their helpful suggestions and feedback. This material is based upon work partially supported by the National Science Foundation under Grants No. 0311808, No. 0448452, No. 0627511, and CCF-0424422, and by the Air Force Office of Scientific Research under MURI Grant No. 22178970-4170. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Air Force Office of Scientific Research, or the National Science Foundation.

References

- [1] Adam Barth, Collin Jackson, and John C. Mitchell. Securing frame communication in browsers. In *Proceedings of the 17th USENIX Security Symposium*, 2008.
- [2] Shuo Chen, David Ross, and Yi-Min Wang. An analysis of browser domain-isolation bugs and a light-weight transparent defense mechanism. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 2–11, New York, NY, USA, 2007. ACM.
- [3] Douglas Crockford. ADsafe.
- [4] Douglas Crockford. ADsafe DOM API.
- [5] Facebook. Facebook Markup Language (FBML).
- [6] Chris Grier, Shuo Tang, and Samuel T. King. Secure web browsing with the OP web browser. In *IEEE Symposium on Security and Privacy*, 2008.
- [7] Jeremiah Grossman. Clickjacking: Web pages can see and hear you, October 2008.
- [8] Norm Hardy. The keykos architecture. *Operating Systems Review*, 1985.
- [9] Norm Hardy. The confused deputy: (or why capabilities might have been invented). *SIGOPS Oper. Syst. Rev.*, 22(4):36–38, 1988.
- [10] Ian Hickson et al. HTML 5 Working Draft.
- [11] Collin Jackson and Helen J. Wang. Subspace: Secure cross-domain communication for web mashups. In *Proceedings of the 16th International World Wide Web Conference. (WWW)*, 2007.
- [12] Peter-Paul Koch. Frame busting, 2004. <http://www.quirksmode.org/js/framebust.html>.
- [13] Butler Lampson. Protection and access control in operating systems. *Operating Systems: Infotech State of the Art Report*, 14:309–326, 1972.
- [14] Sergio Maffei, John C. Mitchell, and Ankur Taly. An operational semantics for JavaScript. In *Proceedings of the 6th Asian Programming Language Symposium (APLAS)*, December 2008.
- [15] Mark Miller. A theory of taming.
- [16] Mark Miller. Caja, 2007.
- [17] Mitre. CVE-2008-4058.
- [18] Mitre. CVE-2008-4059.
- [19] Mitre. CVE-2008-5512.
- [20] Mozilla. XPConnect wrappers. http://developer.mozilla.org/en/docs/XPConnect_wrappers.
- [21] Sape J. Mullender, Guido van Rossum, Andrew Tannenbaum, Robbert van Renesse, and Hans van Staveren. Amoeba: A distributed operating system for the 1990s. *Computer*, 23(5):44–53, 1990.
- [22] Prototype JavaScript framework. <http://www.prototypejs.org/>.
- [23] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. Eros: a fast capability system. In *17th ACM Symposium on Operating System Principles*, New York, NY, USA, 1999. ACM.
- [24] Maciej Stachowiak. Introducing SquirrelFish Extreme, 2008.
- [25] Kris Zyp. CrossSafe.

Physical-layer Identification of RFID Devices

Boris Danev
Dept. of Computer Science
ETH Zürich, Switzerland
boris.danev@inf.ethz.ch

Thomas S. Heydt-Benjamin
IBM Zürich Research
Laboratory, Switzerland
hey@zurich.ibm.com

Srdjan Čapkun
Dept. of Computer Science
ETH Zürich, Switzerland
capkuns@inf.ethz.ch

Abstract

In this work we perform the first comprehensive study of physical-layer identification of RFID transponders. We propose several techniques for the extraction of RFID physical-layer fingerprints. We show that RFID transponders can be accurately identified in a controlled environment based on stable fingerprints corresponding to their physical-layer properties. We tested our techniques on a set of 50 RFID smart cards of the same manufacturer and type, and we show that these techniques enable the identification of individual transponders with an Equal Error Rate of 2.43% (single run) and 4.38% (two runs). We further applied our techniques to a smaller set of electronic passports, where we obtained a similar identification accuracy. Our results indicate that physical-layer identification of RFID transponders can be practical and thus has a potential to be used in a number of applications including product and document counterfeiting detection.

1 Introduction

Passively powered Radio Frequency Identification Devices (RFID) are becoming increasingly important components of a number of security systems such as electronic passports [3], contactless identity cards [4], and supply chain systems [16]. Due to their importance, a number of security protocols have been proposed for RFID authentication [46, 25, 17], key management [31, 28] and privacy-preserving deployment [6, 29, 26, 37, 19, 14, 13]. International standards have been accepted that specify the use of RFID tags in electronic travel documents [3]. Although the literature contains a number of investigations of RFID security and privacy protocols [27, 5] on the logical level, little attention has been dedicated to the security implications of the RFID physical communication layer.

In this work, we focus on the RFID physical communication layer and perform the first study of RFID

transponder physical-layer identification. We present a hardware set-up and a set of techniques that enable us to perform the identification of individual RFID transponders of the same manufacturer and model. We show that RFID transponders can be accurately identified in a controlled measurement environment based on stable fingerprints corresponding to their physical-layer properties. The measurement environment requires close proximity and fixed positioning of the transponder with respect to the acquisition antennas.

Our techniques are based on the extraction of the modulation shape and spectral features of the signals emitted by transponders when subjected to both well formed reader signals, and to out of specification reader signals. We tested our techniques on a set of 50 RFID smart cards of the same manufacturer and type and show that these techniques enable the identification of individual cards with an Equal Error Rate of 2.43% (single run) and 4.38% (two runs). We further applied our techniques to a smaller set of electronic passports, where we obtained a similar identification accuracy. We also tested the classification accuracy of our techniques, and show that they achieve an average classification error of 0% for a set of classes corresponding to the countries of issuance. We further show that our techniques produce features that form compact and computationally efficient fingerprints. Given the low frequencies of operation of the transponders in our study, the extraction of the fingerprints is inexpensive, and could be performed using a low-cost purpose-built reader.

Although the implications of physical-layer identification of RFID transponders are broad, we believe that the techniques we present can potentially find their use in the detection of cloned products and identity documents, where the (stored) fingerprints of legitimate documents are compared with those of the presented documents. Our experimental setup corresponds to this application in which the transponders are fingerprinted from close proximity and in a controlled environment.

It has been recently shown that despite numerous protections, RFIDs in current electronic documents can be successfully cloned [18, 34, 33, 47], even if they apply the full range of protective measures specified by the standard [3], including active authentication. We see our techniques as an additional, efficient and inexpensive mechanism that can be used to detect RFID cloning. More precisely, to avoid detection of a cloned document, an adversary has to produce a clone using a transponder with the same fingerprint as the original document. Although, it may be hard to perform such task, the amount of effort required is an open research problem. We discuss two methods of applying RFID physical-layer identification to cloning detection and compare it to other anti-cloning solutions, like those based on physically-unclonable functions (PUFs) [12].

Our results show the feasibility of RFID transponder fingerprinting in a controlled environment. Using the proposed methods is not enough to extract the same or similar fingerprints from a larger distance (e.g., 1 meter). In our experiments, such remote feature extraction process resulted in incorrect identification. Therefore, we cannot assert that chip holder privacy can be compromised remotely using our techniques. This result further motivates an investigation of physical-layer features of RFID transponders that would allow their remote identification, irrespective of (e.g., random) protocol-level identifiers that the devices use on the logical communication level. Our current results do not allow us to conclude that such distinguishable features can be extracted remotely.

The remainder of this paper is organized as follows. In Section 2, we present our system model and investigation parameters. In Section 3, we detail our fingerprinting setup (i.e., a purpose-built reader), signal capturing process and summarize the data acquisition procedure and collected data. The proposed features for transponder classification and identification are explained in Section 4 and their performance is analyzed in Section 5. We discuss an application of our techniques to document counterfeiting detection in Section 6, make an overview of background and related work in Section 7 and conclude the paper in Section 8.

2 Problem and System Overview

In this work, we explore physical-layer techniques for detection of cloned and/or counterfeit devices. We focus on building physical-layer fingerprints of RFID transponders for the following two objectives:

1. RFID transponder classification: the ability to associate RFID transponders to previously defined transponder classes. In the case of identity docu-

ments classes might, for example, be defined based on the country that issued the document and the year of issuance.

2. RFID transponder identification: the ability to identify same model and manufacturer RFID transponders. In the case of identity documents, this could mean identifying documents from the same country, year and place of issuance.

A classification system must associate unknown RFID transponder fingerprints to previously defined classes C . It performs "1-to- C " comparisons and assigns the RFID fingerprint to the class with the highest similarity according to a chosen similarity measure (Section 5.1). This corresponds to a scenario in which an authority verifies whether an identity document belongs to a claimed class (e.g., country of issuance).

An identification system typically works in one of two modes: either identification of one device among many, or verification that a device's fingerprint matches its claimed identity [8]. In this work, we consider verification of a device's claimed or assumed identity. This corresponds to a scenario in which the fingerprint of an identity document (e.g., passport), stored in a back-end database or in the document chip, is compared to the measured fingerprint of the presented document. The verification system provides an Accept/Reject decision based on a threshold value T (Section 5.1). Identity verification requires only "1-to-1" fingerprint comparison and is therefore scalable in the number of transponders.

In this study we use a single experimental setup for examination of both classification and identification. Our setup consists of two main components: a signal acquisition setup (i.e., a purpose-built RFID reader) (Section 3) and a feature selection and matching component (Section 4). In our signal acquisition setup we use a purpose-built reader to transmit crafted signals which then stimulate a response from the target RFID transponders. We then capture and analyze such responses. In particular, we consider transponder responses when subjected to the following signals from the reader: standard [4] transponder wake-up message, transponder wake-up message at intentionally out-of-specification carrier frequencies, a high-energy burst of sinusoidal carrier at an out-of-specification frequency, and a high-energy linear frequency sweep.

To evaluate the system accuracy, we make use of two different device populations (Table 1). The first population consists of 50 "identical" JCOP NXP 4.1 smart cards [2] which contain NXP RFID transponders (ISO 14443, HF 13.56 MHz). We chose these transponders since they are popular for use in identity documents and access cards, and because they have also been used by hackers to demonstrate cloning attacks against

e-passports [47]. The second population contains 8 electronic passports from 3 different countries¹. These two populations allow us to define different transponder classes (e.g., 3 issuing countries, and a separate class for JCOP cards) for classification and include a sufficient set of identical transponders to quantify the identification accuracy of the transponders of the same model and manufacturer.

In summary, in this work, we answer the following interrelated questions:

1. What is the classification accuracy for different classes of transponders, given the extracted features?
2. What is the identification accuracy for transponders of the same model and manufacturer, given the extracted features?
3. How is the classification and identification accuracy affected by the number of signals used to build the transponder fingerprint?
4. How stable are the extracted features, across different acquisition runs and across different transponder placements (relative to the reader)?

3 Experimental Setup and Data

In this section, we first describe our signal acquisition setup. We then detail the different types of experiments we performed and present the collected datasets from our population of transponders.

3.1 Hardware Setup

Figure 1 displays the hardware setup that we use to collect RF signals from the RFID devices. Our setup is essentially a purpose-built RFID reader that can operate within the standardized RFID communication specifications [4], but can also operate out of specifications, thus enabling a broader range of experiments. The setup consists of two signal generators, used for envelope generation (envelope generator) and for signal modulation (modulation generator), and of transmitting and acquisition antennas. The envelope generator is fed with a waveform that represents the communication protocol wake-up command² required for initiating communication with RFID transponders. The envelope waveform

¹The small quantity of the electronic passports used in the experiments is due to the difficulty of finding people who are in possession of such passports and at the same time willing to allow experimentation on them.

²ISO/IEC 14443 for RFID communication defines two different communication protocols, Type A and B, which use different wake-up commands: WUQA and WUQB, respectively.

is then sent to the modulation generator and is modulated according to the ISO/IEC 14443 protocol Type A or B, depending on the transponders being contacted. The modulated signal is then sent over a PCB transmitting antenna. Finally, the wake-up signal and the response from the transponder are received at the acquisition antenna and captured at the oscilloscope. The separation of the envelope generation and modulation steps allowed us to independently vary envelope and modulation characteristics in our experiments.

In order to collect the RF signal response, we built a "sandwich" style antenna arrangement (Figure 2b) where an acquisition antenna is positioned between the transmission antenna and the target RFID transponder. A wooden platform holds the transmission and acquisition antennas in a fixed position to avoid changes in antenna polarization³. The platform is separated from the desk by a non-metallic wooden cage. The transmission and acquisition antennas are both connected to an oscilloscope. We used the RF signal on the transmission antenna to trigger the acquisition and then record the transponder's response at the acquisition antenna. It should be noted that we can also observe the transponder's response at the transmission antenna, however as the acquisition antenna had a higher gain than the transmission antenna, we used the described setup to obtain better signal-to-noise ratio.

3.2 Performed Experiments

Using the proposed setup, we performed four major experiments:

Experiment 1 (Standard): In this experiment we initiate communication with the transponders as defined by Type A and B protocols in the ISO/IEC 14443 standard. The envelope generator generates the Type A and B envelopes and the modulation generator modulates the signal at a carrier frequency $F_c = 13.56$ MHz, using 100% ASK for Type A and 10% ASK for Type B at the nominal bit rate of $F_b \sim 106$ kbit/s.⁴ The experiment consists of the following steps: a period of unmodulated carrier is transmitted to power the transponder at which time the oscilloscope begins recording the data. The carrier is then modulated according to the envelope such that it corresponds to a WUQA (Type A) or WUQB (Type B) wake-up command. When the commands are no longer transmitted, an unmodulated period of carrier is then sustained while the oscilloscope records the response from the transponder. The carrier is turned off between each

³It has been observed that such changes can reduce the identification accuracy [11].

⁴For 100% ASK modulation we used pulse modulation as standard built-in amplitude modulation (AM) in our generators could not reach the required precision.

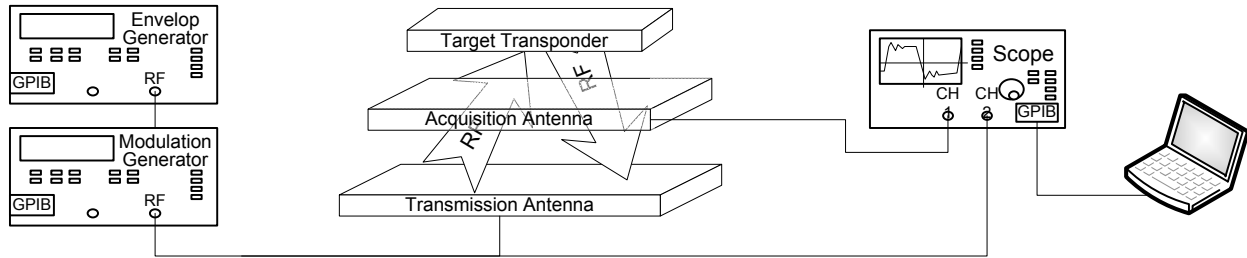


Figure 1: Signal acquisition setup. Envelope and modulation generators generate wake-up signals that initiate the response from the RFID transponder. This wake-up signal is transmitted by the transmitting antenna. The acquisition antenna captures both the wake-up signal and the response from the transponder. The signal from the acquisition antenna is then captured and recorded by the oscilloscope.

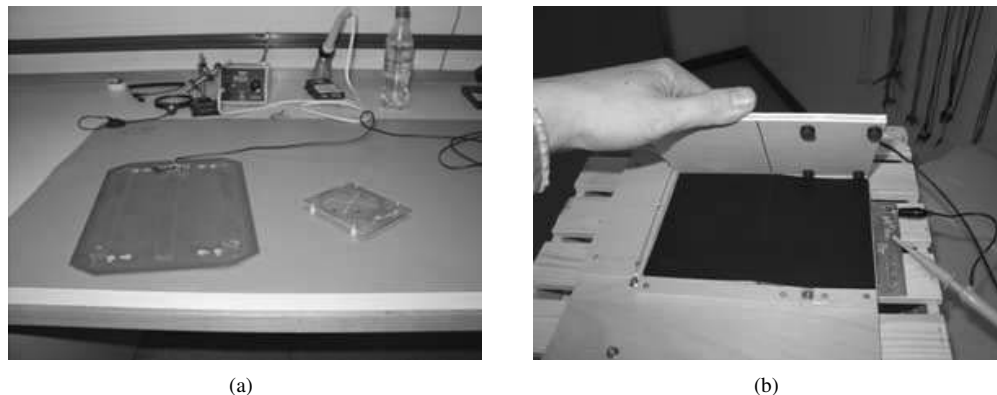


Figure 2: a) Transmission and acquisition antennas. b) An electronic identity document being placed in the finger-printing setup.

observation to ensure the transponder reboots each time. Figures 3a and 3b show the collected samples from Type A and Type B RFID transponders, respectively. This experiment enables us to test if the transponder's responses can be distinguished when they are subjected to standard signals from the reader.

Experiment 2 (Varied F_c): In this experiment, we test transponder responses to the same signals as in Experiment 1, but on out of (ISO/IEC 14443) specification carrier frequencies. Instead of on $F_c=13.56$ MHz, our setup transmits the signals on carrier frequencies between $F_c=12.96$ MHz and 14.36 MHz. Figures 3c and 3d display sample transponder responses to signals on $F_c=13.06$ MHz. We expect the variation in the transponder responses to be higher when they are subjected to out of specification signals, since the manufacturers mainly focus on transponder responses within the specified frequency range.

Experiment 3 (Burst): In this experiment, we tested transponder responses to bursts of RF energy. We subjected the transponders to 10 cycles ($2 \mu s$) of non-modulated 5 MHz carrier at an amplitude of $V_{pp}=10$ V (the maximum frequency and amplitude supported by

our generators while in burst mode). Figure 4a shows a sample transponder response to such an RF burst. This experiment tests the response of transponders to an additional out-of-specification signal. We expect to see variation in different transponders' responses for a variety of reasons. For example since each transponder's antenna and charge pump is unique, we believe that during power-up it will present a unique modulation of an activating field.

Experiment 4 (Frequency Sweep): This experiment consists of observing transponder responses to a non-modulated carrier linear sweep from 100 Hz to 15 MHz at an amplitude of $V_{pp}=10$ V (as measured at transmitting antenna). The duration of the sweep is fixed to the maximum allowed by our generator, 10 ms. In this test we examine how the transponders react to many different frequencies. Among other things, such an experiment provides information about the resonances of the RF circuitry in each transponder. Figure 4b shows a sample transponder response to a frequency sweep. Note the different shape artifacts.

We found that samples collected from Experiment 2 were well suited for transponder classification, whereas

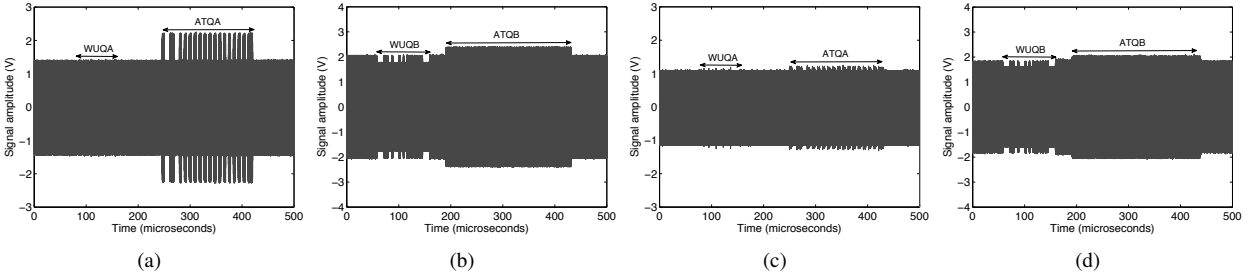


Figure 3: Experiment 1: Type A (a) and Type B (b) RFID transponder responses to WUQA and WUQB commands sent on the ISO/IEC 14443 specified carrier frequency ($F_c=13.56$ MHz). Experiment 2: Type A (c) and Type B (d) RFID transponder responses ATQA and ATQB to WUQA and WUQB commands respectively sent on an out of ISO/IEC 14443 specification carrier frequency ($F_c=13.06$ MHz)

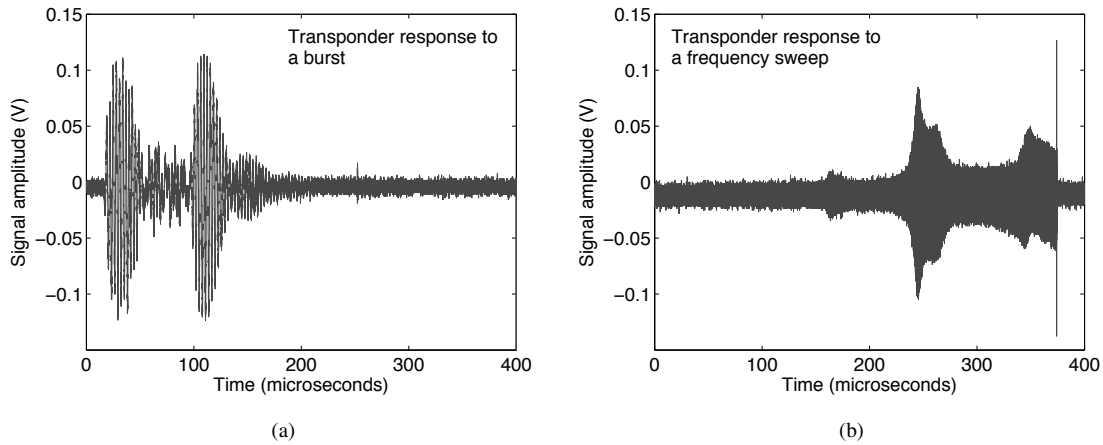


Figure 4: a) Experiment 3: transponder response sample to a non-modulated 5 MHz carrier in duration of 10 cycles. b) Experiment 4: transponder response sample to a non-modulated carrier linear sweep from 100 Hz to 15 MHz. The duration of the sweep is 10 ms.

those collected from Experiments 3 and 4 were better suited for identification of individual RFID transponders. We discuss this result at greater length in Section 4.

3.3 Collected Data

Using the proposed setup, we performed the experiments described in Section 3.2 and collected samples from 8 passports and 50 JCOP NXP 4.1 smart cards (same model and manufacturer). The types of devices used in the experiments are shown in Table 1. For the privacy of our research subjects we arbitrarily labeled the passports as ID1 to 8. To further protect their privacy we give the country and place of issuance under the pseudonyms C1 to C3 and P1 to P6 respectively.

Our data collection procedure for a single experiment "run" was as follows: We positioned the target RFID device on the experimental platform with all other transponders being at an out-of-range distance from the

activating field. We then placed a heavy non-metallic weight on top of the transponder to position it firmly and horizontally on the platform. For each device we then performed Experiments 1-4 at fixed acquisition timing offset and sampling rate and saved the samples to a disk for later analysis. For each transponder we performed two runs, completely removing and replacing the target transponder on the experimental platform between runs. This ensures that extracted features are stable across repositioning.

In each iteration of Experiment 2 we collected 4 samples per run for 14 different carrier frequencies starting from $F_c=12.96$ up to 14.36 MHz with a step of 100 KHz. This resulted in 64 samples per transponder per run. In Experiments 3 and 4 we collected 50 samples per device per run.

Table 1: RFID device populations (passports and JCOP NXP smart cards).

Type	Number	Label	Country	Year	Place of Issue
Passport	2	ID1, ID2	C1	2006	P1
	1	ID3	C1	2006	P2
	1	ID4	C1	2006	P3
	1	ID5	C1	2007	P4
	1	ID6	C2	2008	P5
	1	ID7	C3	2008	P6
	1	ID8	C1	2008	P1
JCOP	50	J1..J50	JCOP NXP 4.1 cards (same model and manufacturer)		

4 Feature Extraction and Selection

The goal of the feature extraction and selection is to obtain distinctive fingerprints from raw data samples collected in the proposed experiments, which most effectively support the two objectives in our work, namely classification and identification. In this section, we detail the extraction and matching procedures of two types of features effective for that purpose: modulation-shape features (Section 4.1) and spectral PCA features (Section 4.2). We also investigated the use of some timing features, such as the time interval within which the transponder responds to an WUQ command and the duration of that response (Figure 5a). These timing features performed poorly in both tasks, hence in this work we focus on the modulation-shape and spectral features.

4.1 Modulation-shape Features

In this section, we describe the extraction and matching procedures for the features extracted from the shape of the modulated signal of the transponder responses at a given carrier frequency F_c (Experiment 1&2). Figure 5 b) shows the shape of the On-Off keying modulation for the JCOP NXP 4.1 card for the first packet in a transponder’s response to a wake-up command. All Type A transponders in our study had a logically identical first packet.

For a given transponder, the features of the modulated signal are extracted from the captured transponder response (see Figure 3) denoted as $f(t, l)$, using Hilbert transformation. Here, $f(t, l)$ is the amplitude of the signal l at time t . The Hilbert transformation is a common transformation in signal processing used to obtain the signal envelope [38].

In Step (i), we apply Hilbert transformation on $f(t, l)$ to obtain $H(t, l)$:

$$H(t, l) = \text{Hil}(f(t, l)) \quad (1)$$

where Hil is a function implementing the Hilbert transform [36].

In Step (ii), the starting point of the modulation in $H(t, l)$ is determined using the variance-based threshold detection algorithm described in [40]. The end point is fixed to a pre-defined value (see Section 5) and then the modulation-shape fingerprint is extracted.

Feature matching between a reference and a test fingerprints is performed using standardized Euclidean distance, where each coordinate in the sum of squares is inverse weighted by the sample variance of that coordinate [35].

4.2 Spectral Features

In this section, we describe the extraction and matching of spectral features from data collected from Experiments 3 (Burst) and 4 (Sweep) (Section 3.2).

Both frequency sweep and burst data samples are extremely high-dimensional: each sweep data sample contains 960000 points (dimensions) and each burst data sample contains 40000. Such high-dimensional data typically contain many noisy dimensions which are detrimental to finding distinctive features. Therefore, it is critical to remove the noise as much as possible from the raw data samples.

We explored two basic approaches to solve the dimensionality problem. In the first approach, we considered transforming the data in the frequency domain by means of the Fast Fourier Transform (FFT) and remove the high frequencies (usually considered noisy) by filtering. However, matching experiments using direct vector similarity measures such as Euclidean and Cosine distance failed to produce distinctive enough features. This may be because in removing the high frequencies we are also removing frequencies that contain discriminative capabilities. Such behavior is commonly noticed in biometrics research [10]. In the second approach we down-sampled the signal at different rates in order to reduce the dimensionality. We then transformed the data in the frequency domain by FFT and applied standard vector similarity measures. Again reducing the dimensionality in this way did not prove to be effective in extracting sufficiently dis-

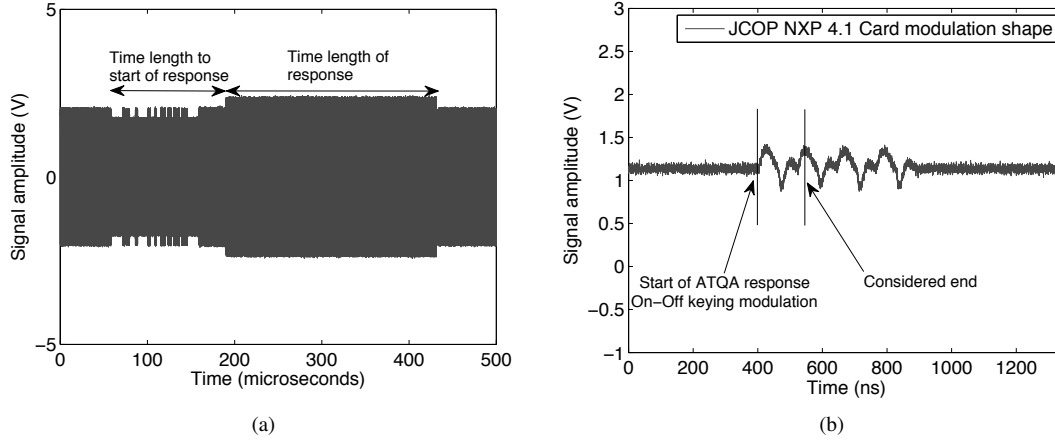


Figure 5: a) Timing features extracted from Type B transponder responses. b) Modulation-shape features.

criminative features.

To overcome the above problems, we use a modification of Principal Component Analysis (PCA) for high-dimensional data [7], that reduces data dimensionality by discarding dimensions that do not contribute to the total covariance. Given that the number of dimensions is very high, orders of magnitude higher than the number of data samples we can process, a standard PCA procedure cannot be applied. In the following subsection, we briefly describe the used PCA modification.

4.2.1 Feature Extraction and Matching

For a given RFID device, spectral PCA features are extracted from N captured samples using a linear transformation derived from PCA for high-dimensional data. We denote a signal by $f(t, l)$, where $f(t, l)$ is the amplitude of the signal l at time t . The features are extracted in the following three steps:

In Step (i), we apply a one-dimensional Fourier transformation on $f(t, l)$ to obtain $F(\omega, l)$:

$$F(\omega, l) = \frac{1}{\sqrt{M}} \sum_{m=0}^{M-1} f(t, l) \exp(-2\pi i \frac{t\omega}{M}) \quad (2)$$

where M is the length of signal considered and $0 \leq t \leq M - 1$ is time. We then remove from $F(\omega, l)$ its DC component and the redundant part of the spectrum; we denote the remaining part of the spectrum by \vec{s}_l .

In Step (ii), a projected vector \vec{g}_l , also called a spectral feature, is extracted from the Fourier spectrum using a PCA matrix W_{PCA} :

$$\vec{g}_l = W_{PCA}^t \vec{s}_l \quad (3)$$

The feature extraction from N captured samples for a given transponder is then given by $G = W_{PCA}^t S$ where G is an array of \vec{g}_l and S is a matrix $S = [\vec{s}_0 \dots \vec{s}_l \dots \vec{s}_N]$.

Finally, in Step (iii), the feature template (fingerprint) h used for matching is computed:

$$h = \{\hat{G}; \Sigma_G\} \quad (4)$$

where \hat{G} denotes the mean vector of G and Σ_G denotes the covariance matrix of G . The number of captured samples N used to build the feature template and the number of projected vectors in W_{PCA} (i.e., the subspace dimension) are experimentally determined.

Mahalanobis distance is used to find the similarities between fingerprints⁵. The result of matching a reference h^R and a test h^T feature templates is a matching score, calculated as follows.

$$scr(h^R, h^T) = \min(\sqrt{(\hat{G}^T - \hat{G}^R)^t \Sigma_G^{-1} (\hat{G}^T - \hat{G}^R)}, \sqrt{(\hat{G}^T - \hat{G}^R)^t \Sigma_{G^T}^{-1} (\hat{G}^T - \hat{G}^R)}) \quad (5)$$

Values of the matching score closer to 0 indicate a better match between the feature templates. The proposed matching uses the mean and covariance of both test and reference templates. It also ensures the symmetric property, that is $scr(h^R, h^T) = scr(h^T, h^R)$.

It should be noted that the proposed feature extraction and matching method can be efficiently implemented in hardware as they use only linear transformations for feature extraction and inter-vector distance matchings. These operations have a low memory footprint and are computationally efficient.

4.2.2 PCA Training

In order to compute the eigenvalues and corresponding eigenvectors of the high-dimensional data (the number

⁵We discovered that the feature templates are distributed in ellipsoidal manner and therefore use Mahalanobis distance that weights each projected sample according to the obtained eigenvalues.

of samples \ll the number of dimensions), we used the following lemma:

Lemma: For any $K \times D$ matrix W , mapping $x \rightarrow Wx$ is a one-to-one mapping that maps eigenvectors of $W^T W$ onto those of $W W^T$.

Here W denotes a matrix containing K samples of dimensionality D . Using this lemma, we can first evaluate the covariance matrix in a lower space, find its eigenvectors and eigenvalues and then compute the high-dimensional eigenvectors in the original data space by normalized projection [7]. Based on this description, we compute the PCA matrix $W_{PCA} = [\vec{u}_1 \vec{u}_2 \dots \vec{u}_i]$ by solving the eigenvector equation:

$$\left(\frac{1}{K} X^T X\right)(X^T \vec{v}_i) = \lambda_i (X^T \vec{v}_i) \quad (6)$$

where X is the training data matrix $K \times D$ and \vec{v}_i are the eigenvectors of XX^T . We then compute the eigenvectors of our matrix \vec{u}_i by normalizing:

$$\vec{u}_i = \frac{1}{\sqrt{K\lambda_i}} (X^T \vec{v}_i) \quad (7)$$

It should be noted that other algorithms like probabilistic PCA (e.g., EM for PCA) can potentially be also used given the fact that we discovered that only 5-10 eigenvectors are predominant. We intend to investigate these as a part of our future work.

5 Performance Results

In this section, we present the performance results of our fingerprinting system. First, we review the metrics that we use to evaluate the classification and identification accuracy.

5.1 Evaluation Metrics

As a metric for classification, we adopt the average classification error rate, defined as the percentage of incorrectly classified signatures to a predefined set of classes of signatures (e.g., countries). We used the 1-Nearest Neighbor rule [7] for estimating the similarity between testing and reference signatures from a given class; that is, a testing signature is matched to all reference signatures from all classes and assigned to the class with nearest distance similarity. It should be noted that more sophisticated classifiers can be devised such as Support Vector Machines (SVM), Probabilistic Neural Networks (PNN) [7]. However these classifiers require more training which we do not consider in this work.

As metrics for identification, we adopt the Equal Error Rate (EER) and the Receiver Operating Characteristic (ROC) since these are the most agreed metrics for evaluating identification systems [8]. The False Accept Rate

(FAR) and the False Reject Rate (FRR) are the frequencies at which the false accept and the false reject events occur. The FAR and FRR are closely related to each other in the Receiver Operating Characteristic (ROC). ROC is a curve which allows to automatically compute FRR when the FAR is fixed at a desired level and vice versa [8]. The operating point in ROC, where FAR and FRR are equal, is called the Equal Error Rate (EER). The EER represents the most common measure of the accuracy of identification systems [1]. The operating threshold value at which the EER occurs is our threshold T for an Accept/Reject decision.

To increase the clarity of presentation, we use the Genuine Accept Rate ($GAR = 1 - FRR$) in the ROC because it shows the rate of Accepts of legitimate identities. In addition, we also compute FRR for common target values of FAR (e.g., $FAR = 1\%$).

5.2 Classification Results

In this section, we present the results of the classification using modulation-shape and spectral features. In this evaluation, we consider all our passport samples and 5 of the JCOP NXP 4.1 cards. Here, the identity documents ID1, ID2, ID3, ID4, ID7, ID8 (see Table 1) and the JCOP cards implement Type A communication protocol, whereas ID5 and ID6 use Type B protocol. It is interesting to notice that within the same country class (C1) we have documents with two different communication protocols (ID1-ID4 and ID8 implement Type A, whereas ID5 implements Type B protocol).

5.2.1 Classification using Modulation-shape Features

The modulation-shape features described in Section 4 show the discriminant artifacts in the transponder's response. In particular, we discovered that these artifacts (shapes) vary from one transponder to another on out-of-specification carrier frequencies.

Figure 6 shows the modulation envelope shapes of the initial sequence of the RFID transponder's response after Hilbert transformation for 4 different classes of Type A protocol devices. These were recorded at an out of specification carrier frequency $F_c = 13.16\text{MHz}$. Visual inspection shows that the modulation shapes not only differ from class to class but also are stable within different runs.

In order to quantify these observations more precisely, we considered classification with 3 classes (2 countries + JCOP cards) with all fingerprints from two different runs. The classification process was repeated 8 times with 8 different reference fingerprints per class for validation.

Table 2: Classification using modulation-shape features (Experiment 2)

Number of Classes	Class structure	Average Classification Error Rate
3	(C1),(C2),(JCOP)	0%
4	(ID1,ID3,ID4,ID8), (ID2), (ID7), (JCOP)	0%
2	(ID5-C1),(ID6-C3)	0%

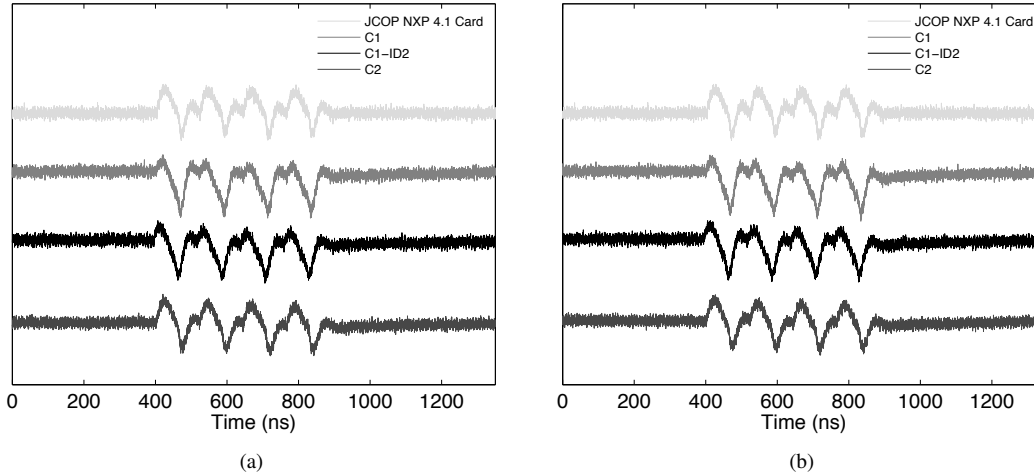


Figure 6: Modulation shape of the responses of 4 different classes (C1),(C1-ID2),(C2),(JCOP): a) first run b) second run. In each run, the sample transponders were freshly placed in the fingerprinting setup. These plots show the stability of the collected modulation-shape features across different runs.

The results show perfect separability of the classes with average classification error rate of 0%. In addition, after detailed inspection of the modulation-shape features we discovered that ID2 from C1 differs significantly from the representatives of that class. We therefore formed a new classification scenario with 5 classes and obtained again a classification error rate of 0%. It is an interesting result given that ID1 and ID2 are issued by the same country, in the same year and place of issue. However, their transponders are apparently different. The modulation-shapes of ID1, ID3 and ID4 from C1 could not be further distinguished using the combination of modulation-shape features and Euclidean matching. Table 2 shows the results.

Similar to Type A, the 2 Type B transponders from two different countries (C1,C3) available in our population showed complete separability with classification error rate of 0%. We acknowledge that our data set is insufficient due to the difficulty of obtaining e-passports. We believe however that our results are promising to stimulate future work with a larger set of e-passports.

In summary, the modulation shapes at an out-of-specification carrier frequency are successful in categorizing different classes of transponders (e.g., countries). They are quickly extractable and stable across different runs. For the classification task, there is no need of statis-

tical analysis in contrast with the proposed spectral features analyzed in the next sections. An additional advantage is that specialized hardware is not required as current RFID readers can be easily adapted.

5.2.2 Classification using Burst and Sweep Spectral Features

We also performed classification using burst and sweep spectral features (Experiment 3 & 4) on the same set of classes as with modulation-shape features (Table 2). Similar to the modulation-shape features, this classification achieved a 0% classification error rate on the proposed classes. Moreover, using the spectral features we were also able to distinguish individually each of our 9 identity documents with an EER=0%, i.e. we were able to verify the identity of each individual document with an accuracy of 100% with FRR=FAR=0%. This result motivated us to estimate the identification accuracy of spectral features on a larger set of identical (of the same make and model) transponders.

5.3 Identification results

In this section we present the results of the identification capabilities of the (burst and sweep) spectral features for

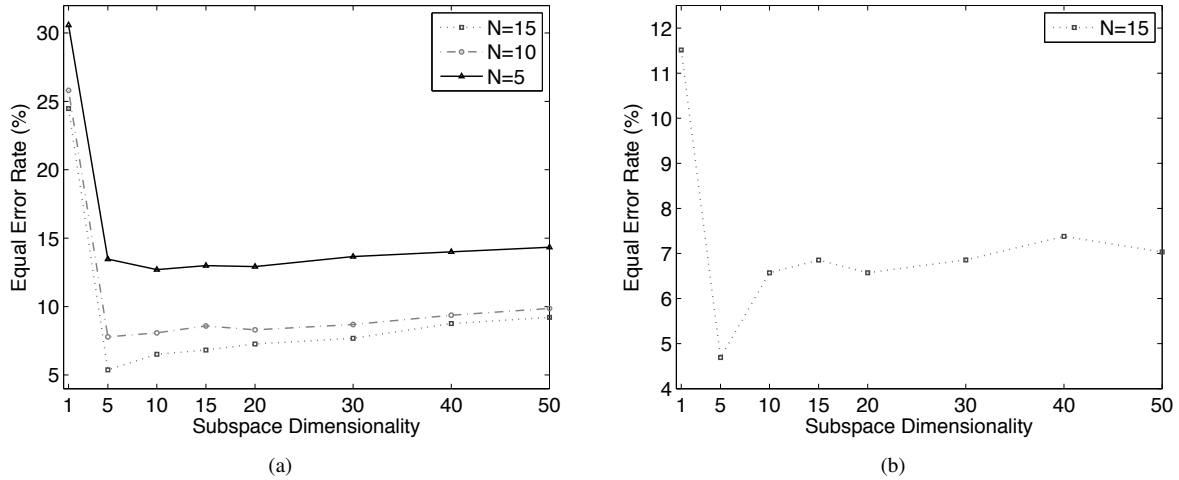


Figure 7: Spectral features identification accuracy for different number of samples N used to build the fingerprint and for different subspace dimensions: a) burst spectral features, b) sweep spectral features. 50 identical (same manufacturer and model) transponders are used in the computation.

our data population (50 identical JCOP NXP 4.1 cards). We adopt the following approach. We first evaluate the accuracy over the data collected in a single run of the experiment (Section 5.3.1 and 5.3.2). We then quantify the feature stability of the spectral features by considering samples from two independent runs together (Section 5.3.3).

We validate our results using cross-validation [7]. We measured 50 samples per transponder per run of which we use 5-10 samples for training and the remaining 40-45 samples for testing depending on the number of samples N used to build the fingerprint. The training and testing data are thus separated and allow validation of the identification accuracy.

5.3.1 Identification using Burst Spectral Features

In this evaluation, we consider the samples from the burst dataset, from a single experiment run (Experiment 3) in order to obtain a benchmark accuracy. We varied two parameters: the number of samples N used to build the feature templates (fingerprints) and the dimension of the PCA subspace used to project the original features into. The dimension of the PCA subspace is also related to the feature template size which we discuss below.

The results of this analysis are presented in Figure 7a for different N and subspace dimensionality. The dimension of the features before the projection is 19998. The results show the EER of the system reaching 0.0537 (5.37%) for $N=15$. This means that our system correctly identifies individual identical transponders with an accuracy of approximately 95% (GAR at the EER operating point) using the features extracted from the burst sam-

ples. We later show that this accuracy is preserved in cross-matchings between different runs. Table 3 summarizes the underlying data, namely the number of samples N , total genuine and imposter matchings performed for EER computation⁶, Accept/Reject threshold, EER and confidence interval (CI).

The results in Figure 7a also confirm that using the first 5 eigenvectors to project and store the feature template provides the highest accuracy. Our proposed features therefore form compact and computationally efficient fingerprints (see Section 5.4).

5.3.2 Identification using Sweep Spectral Features

Similarly to the above analysis, we considered the first run of samples from the sweep experiment (Experiment 4) dataset. For computational reasons, we did not consider the entire sample. Instead, we extracted the spectral features from the part of the sample between 220 to 270 microseconds. As it can be seen in Figure 4, this part contains the biggest shape changes in the frequency sweep. This decision reduced the considered space to 100000 points which allowed reasonably fast feature extraction (26 s per sample). This clearly excludes some discriminant information from our analysis, and future work should include other sections of the sample signals.

The results are presented in Figure 7b for $N=15$ and

⁶The number of genuine and imposter matchings depends on the number of available fingerprints per transponder. For $N=10$, we are able to build 4 different fingerprints with the testing data within a run. This results in 6 different matchings of fingerprints from the same device (i.e., genuine matchings) and 392 different matchings of fingerprints from different transponders (i.e., imposter matchings). For 50 transponders, this makes 300 genuine and 19600 imposter matchings.

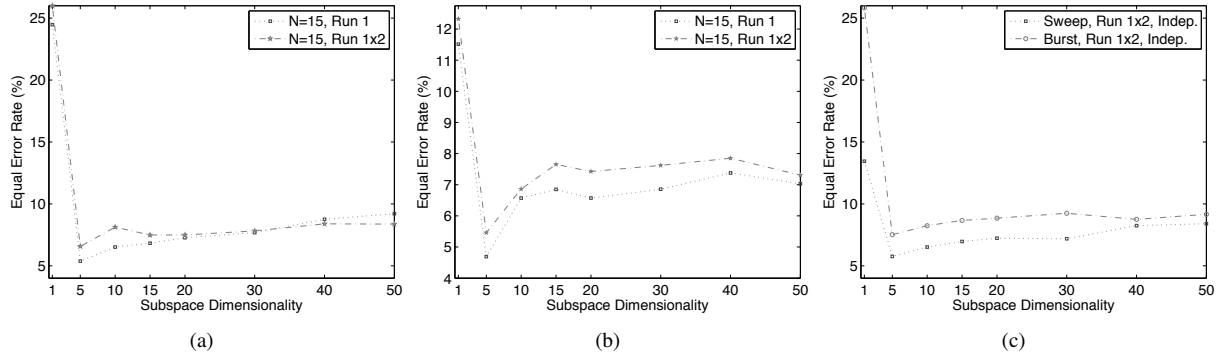


Figure 8: Feature stability in identification: a) burst spectral features b) sweep spectral features. 50 identical (same manufacturer and model) transponders are used in the experiments. c) burst and sweep spectral features on independent transponder sets for training and testing; 20 transponders are used for training and 30 transponders - for testing; $N=15$.

different subspace dimensions. The dimension of the original features before projection is 49998. We computed the EER for $N=15$ (see Burst analysis in Section 5.3.1). The obtained EER is 0.0469 (4.69%), when using the first 5 eigenvectors to project and store the feature template. The obtained accuracy is therefore similar to the one obtained with the burst features, i.e. our system correctly identifies the individual identical transponders with an accuracy of approximately 95% (GAR at the EER point). Table 3 shows the confidence intervals.

5.3.3 Feature Stability

In the previous sections we have analyzed the identification accuracy using burst and sweep spectral features within a single experiment run. This allows us to have a benchmark for estimating the stability of the features. In particular, we performed the following stability analysis:

1. Using the linear transformations W_{PCA} obtained in the first run, we selected 4 feature templates (2 from each run) and computed again the EER by considering only the cross matching scores of fingerprints from different runs⁷. The process was repeated 3 times with different feature templates from the two runs to validate the feature stability.
2. We trained the system over the first 20 transponders and then used the obtained linear transformation to estimate the accuracy over the remaining 30 transponders. This analysis tests the stability of the obtained linear transformations to discriminate independent transponder populations⁸.

⁷This procedure is required in order to remove any possible bias from cross matching scores of fingerprints from the same run. We point out that this results in a reduced number of genuine and imposter matchings for the EER computation, 200 and 9800 respectively (see Table 3).

⁸The motivation behind this division (20 vs. 30) is that it gives

Figure 8 compares the EER accuracy obtained with the first run (Run 1) and the accuracy obtained by mixing fingerprints of both runs (Run 1×2) for a fixed $N=15$. Table 3 displays the confidence interval for subspace dimension of 5 eigenvectors. The obtained EERs do not show a statistically significant difference between the two experiments for both the burst and sweep features using 4-fold validation.

Figure 9 displays the EER accuracy obtained using independent transponder sets for training and testing for a fixed $N=15$. Here, the fingerprints from both runs are mixed as in the previous analysis. Table 4 summarizes the numeric results together with confidence intervals of the EER. Even though the testing population (30 transponders) is smaller, we observe that the sweep features do not show any significant accuracy deviation from the benchmark accuracy on Run 1×2 (Table 3). On the other hand, the burst features slightly decreased the accuracy on average (Table 3). The reason for this might be that 20 different transponders are not sufficient to train the system; however, we cannot assert this with certainty.

5.3.4 Combining Sweep and Burst Features

Given that the identification accuracies of both burst and sweep spectral features are similar; in order to fully characterize the identity verification we computed the ROC curves for the burst and sweep features as shown in Figure 9b. We notice that while the EERs are similar, the curves exhibit different accuracies at different FARs. In particular, for low $FAR \leq 1\%$ the sweep features show lower GAR.

The burst and sweep features discriminate the fingerprints in a different way, and therefore these features can be combined in order to further increase the accuracy. Such combinations are being researched in multi-modal

reasonable number of transponders for both training and testing.

Table 3: Summary of accuracy for the 5-dimensional spectral features (50 transponders).

Type	Run	N	Test matchings		Threshold T	EER (%)	EER CI (%)		Validation
			Genuine	Imposter			lower	upper	
Burst	1	15	150	11025	1.88	5.37	4.38	6.36	4-fold
	1	10	300	19600	2.91	7.79	5.29	10.28	4-fold
	1	5	300	19600	7.56	13.47	13.22	13.72	4-fold
	1x2	15	200	9800	2.64	6.57	6.25	6.89	4-fold
Sweep	1	15	150	11025	1.68	4.69	3.65	5.74	4-fold
	1x2	15	200	9800	1.93	5.46	5.08	5.84	4-fold

Table 4: Accuracy when independent sets are used for training (20) and testing (30) transponders.

Type	Run	N	Test matchings		Threshold T	EER (%)	EER CI (%)		Validation
			Genuine	Imposter			lower	upper	
Burst	1x2	15	120	3480	2.78	7.33	6.01	8.65	3-fold
Sweep	1x2	15	120	3480	2.03	5.75	5.45	6.05	3-fold

biometrics [42] where different "modalities" (e.g., fingerprint and vein) are combined to increase the identification accuracy and bring more robustness to the identification process [42].

A number of integration strategies have been proposed based on decision rules [32], logistic functions to map output scores into a single overall score [24], etc. Figure 9 shows the EERs and ROC curves of feature combination by using the sum as an integration function. The overall matching score between a test and a reference template is the sum of the matching scores obtained separately for the burst and sweep features. Table 5 summarizes the results.

For the benchmark datasets (Run 1), we observe significant improvement of the accuracy reaching an EER=2.43%. The improvement is also significant for all target FARs (e.g., 0.1%, 1%) as shown in Figure 9b. We also observe a statistically significant improvement on using fingerprints from both Run 1 and 2. The accuracy is slightly lower (EER=4.38%). These results motivate further research on feature modalities and novel integration strategies.

5.4 Summary and Discussion

In this section, we have experimentally analyzed the classification and identification capabilities of three different physical-layer features with related signal acquisition, feature extraction and matching procedures.

The results show that classification can successfully be achieved using the modulation shape of the transponder's response to a wake-up command at an out-of-specification frequency (e.g., $F_c=13.06$ MHz). This technique is fast, does not require special hardware and can be applied without statistically training the classification process.

For identification, we proposed using spectral features extracted from the transponder's reaction to purpose-built burst and linear frequency sweep signals. Our proposed signal acquisition and feature extraction/matching techniques achieved separately an identification accuracy of approximately EER=5% over 50 identical RFID transponders. The proposed features are stable across acquisition runs. In addition, our spectral features showed that they can be combined in order to further improve the accuracy to EER=2.43%.

The results also confirm that using the first 5 eigenvectors is sufficient to represent the proposed features while keeping the identification accuracy high. Therefore, our proposed features also form very compact and computationally efficient fingerprints. Typically, if each dimension is represented by a 4-byte floating-point number, the size of the corresponding feature template $h = \{\hat{G}; \Sigma_G\}$ is 20 (5×4) bytes for \hat{G} and 100 ($5 \times 5 \times 4$) bytes for the square covariance matrix Σ_G resulting in a total of 120 bytes.

In terms of feature extraction performance, given the much lower dimensionality of the burst samples (40000 vs. 960000 for the sweep), they are much faster to digitally acquire and extract with approximately 2 sec. compared to 26 sec. for the sweep data samples. The times are measured on a machine with 2.00 GHz CPU, 2 GB RAM running Linux Ubuntu. It should be noted that all the components of the feature extraction can be implemented efficiently in hardware which would significantly improve the performance.

6 Application to Cloning Detection

The classification and identification results presented in Section 5 indicate that physical-layer fingerprinting can be practical in a controlled environment. In this section,

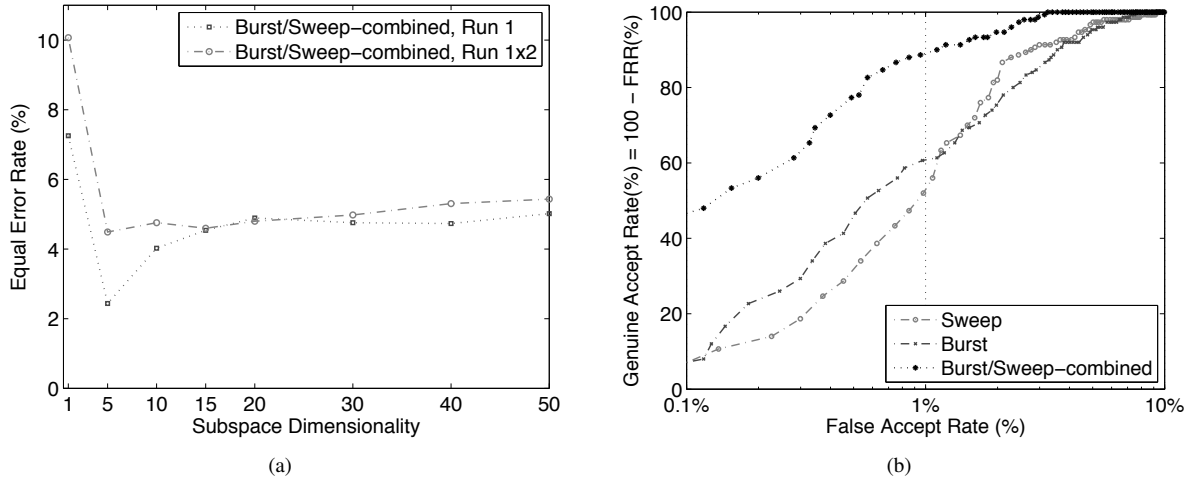


Figure 9: a) The identification accuracy combining the sweep and burst features b) Receiver Operating Characteristic (ROC) for $N=15$ for burst and sweep spectral features and their combination. 50 identical transponders are used. The subspace dimension is fixed to 5. See Table 5 for the underlying data.

Table 5: Summary of accuracy when a combination of burst and sweep features used (50 transponders).

Type	Run	N	Test matchings		Threshold T	EER (%)	EER CI (%)		Validation
			Genuine	Imposter			lower	upper	
Burst/Sweep	1	15	150	11025	1.56	2.43	1.54	3.33	4-fold
Burst/Sweep	1x2	15	200	9800	2.18	4.38	3.9	4.9	4-fold

we discuss how it could be used in the context of product or document cloning detection. We point out however that the cloning detection will obey to the achieved error rates. Despite a number of protective measures, it has been recently shown [18, 34, 33, 47] that even RFID transponders in electronic identity documents can be successfully cloned, even if the full range of protective measures specified by the standard [3], including active authentication, is used. We consider the physical-layer fingerprinting described in this work as an additional efficient mechanism that can be used to detect document counterfeiting.

We foresee two use cases in which fingerprints can be applied for anti-counterfeiting. In the first use case, the fingerprints are measured before RFID deployment and are stored in a back-end database, indexed with the unique transponder (document) identifier. When the authenticity of the document with identifier ID is verified, the fingerprint of the document transponder is measured, and then compared with the corresponding transponder fingerprint of document ID stored in the database. In order to successfully clone the document, the attacker needs to perform two tasks:

1. Obtain the fingerprint template of the transponder in the original document and

2. Produce or find a document (transponder) with the same fingerprint.

In order to extract a fingerprint template the attacker needs to fully control the target document (hold it in possession) for long enough to complete the extraction. Using the methods from our study, it would be hard, if not infeasible, for the attacker to extract the same fingerprints remotely (e.g., from few meters away). In our experiments, such remote feature extraction process resulted in an EER of approximately 50%. We assume that this is due to the change of acquisition antenna orientation and lower signal-to-noise ratio. We do not exclude the possibility that other discriminant features could be found that could be extracted remotely. However, this does not appear to be the case for our features. After obtaining the original fingerprint, the attacker now needs to produce or find an RFID transponder with that fingerprint (i.e., such that it corresponds to the one of the original document), which is hard given that the extracted fingerprints are due to manufacturing process variation. Although manufacturing process variation effects the RFID micro-controller itself, it is likely that the main source of detectable variation lies in the RFID radio circuitry. However, we cannot conclude with certainty which component of the entire transponder circuit contributes most to the fingerprints. We leave this determination to future

work. Because of the complexity of these circuits this is a challenging task in the lab, let alone in "the wild" environment of the attacker.

In the second use case, transponder fingerprints are measured before their deployment as in the first case, but are stored on the transponders instead of in a back-end database. Here, we assume that the fingerprints stored on the transponders are digitally signed by the document-issuing authority and that they are protected from unauthorized remote access; the digital signature binds the fingerprint to the document unique identifier, and both are stored on the transponder. When the document authenticity is validated, the binding between the document ID and the fingerprint stored on the transponder is ensured through cryptographic verification of the authority's signature. If the signature is valid, the stored fingerprint is compared to the measured fingerprint of the document transponder. The main advantage in this use case is that the document authenticity can be verified "off-line". The main drawback is that the fingerprint is now stored on the transponder and without appropriate access protection, it can be remotely obtained by the attacker. Here, minimal access protection can be ensured by means of e.g., Basic Access Authentication [3] although, that mechanism has been shown to have some weaknesses due to predictable document numbers [33]. As we mentioned in Section 5.4, our technique generates compact fingerprints, which can be stored in approximately 120 bytes. This means that they can easily be stored in today's e-passports. The ICAO standard [3] provides space for such storage in files EF.DG[3-14], which are left for additional biometric and future use; transponder fingerprints can be stored in those files. Our proposal does not require the storage of a new public key or maintenance of a separate public-key infrastructure, since the integrity of the fingerprints, stored in EF.DG[3-14] will be protected by the existing passive authentication mechanisms implemented in current e-passports.

The closest work to ours in terms of transponder cloning protection is the work of Devadas et al. [12], where the authors propose and implement Physically Unclonable Function(PUF)-Based RFID transponders. Processors in these transponders are specially designed and contain special circuits, PUFs, that are hard to clone and thus prevent transponder cloning. The main difference between PUF-based solutions and our techniques is that our techniques can be used with existing RFID transponders, whereas PUF-based solutions can detect cloning only of PUF-based transponders. However, PUF-based solutions do have an advantage that they rely on "controlled" randomness, unlike our techniques, that relies on randomness that is unintentionally introduced in the manufacturing of the RFID tags.

7 Related Work

Besides PUF-based RFIDs [12], that we discuss in the previous section, the following works relate to ours.

In [41], Richter et al., report on the possibility of detecting the country that issued a given passport by looking at the bytes that an e-passport sends as a reply in response to some carefully chosen commands from the reader. This technique therefore enables classification of RFID transponders used in e-passports. Our technique differs from that proposal as it enables not only classification, but also identification of individual passports. Equally, the technique proposed in [41] cannot be used for cloning detection since the attacker can modify the responses of a tag on a logical level.

The proliferation of radio technologies has triggered a number of research initiatives to detect illegally operated radio transmitters [44, 45, 23], mobile phone cloning [30], defective transmission devices [48] and identify wireless devices [20, 22, 43, 40, 39, 9] by using physical characteristics of the transmitted signals [15]. Below, we present the most relevant work to ours in terms of signal similarities, features and objectives.

Hall et al. [20, 21] explored a combination of features such as amplitude, phase, in-phase, quadrature, power and DWT of the transient signal. The authors tested on 30 IEEE 802.11b transceivers from 6 different manufacturers and scored a classification error rate of 5.5%. Further work on 10 Bluetooth transceivers from 3 manufacturers recorded a classification error rate of 7% [22]. Ureten et al. [39] extracted the envelope of the instantaneous amplitude by using the Hilbert transformation and classified the signals using a Probabilistic Neural Network (PNN). The method was tested on 8 IEEE 802.11b transceivers from 8 different manufacturers and registered a classification error rate of 2%-4%. Rasmussen et al. [40] explored transient length, amplitude variance, number of peaks of the carrier signal and the difference between mean and maximum value of the transient. The features were tested on 10 identical Mica2 (CC1000) sensor devices (approx. 15cm from the capturing antenna) and achieved a classification error rate of 30%. Brik et al. [9] proposed a device identification technique based on the variance of modulation errors. The method was tested on 100 identical 802.11b NICs (3-15 m from the capturing antenna) and achieved a classification error rate of 3% and 0.34% for k-NN and SVM classifiers respectively. In [11] the authors demonstrate the feasibility of transient-based Tmote Sky (CC2420) sensor device identification with an EER of 0.24%. The same work considered the stability of the proposed fingerprint features with respect to capturing distance, antenna polarization and voltage, and related attacks on the identification system.

8 Conclusion

In this work we performed the first comprehensive study of physical-layer identification of RFID transponders. We showed that RFID transponders have stable fingerprints related to physical-layer properties which enable their accurate identification. Our techniques are based on the extraction of the modulation shape and spectral features of the response signals of the transponders to the in- and out- of specification reader signals. We tested our techniques on a set of 50 RFID smart cards of the same manufacturer and type and we showed that these techniques enable the identification of individual transponders with an Equal Error Rate of 2.43% (single run) and 4.38% (two runs). We further applied our techniques to a smaller set of electronic passports, where we obtained a similar identification accuracy. We tested the classification accuracy of our techniques, and showed that they achieve 0% average classification error for a set of classes corresponding to manufacturers and countries of issuance. Finally, we analyzed possible applications of the proposed techniques to the detection of cloned products and documents.

Acknowledgements

This work was partially supported by the Zurich Information Security Center. It represents the views of the authors.

References

- [1] Fingerprint verification competitions (FVC). <http://bias.csr.unibo.it/fvc2006/>.
- [2] IBM JCOP family. <ftp://ftp.software.ibm.com/software/pervasive/info/JCOP.Family.pdf>.
- [3] ICAO. <http://www.icao.int/>.
- [4] ISO/IEC 14443 standard. <http://www.iso.org/>.
- [5] RFID security and privacy lounge. <http://www.avoine.net/rfid/index.html>.
- [6] AVOINE, G., AND OECHSLIN, P. RFID traceability: A multi-layer problem. In *Financial Cryptography* (2005), A. Patrick and M. Yung, Eds., vol. 3570 of *LNCS*, pp. 125–140.
- [7] BISHOP, C. *Pattern Recognition and Machine Learning*. Springer, 2006.
- [8] BOLLE, R., CONNELL, J., PANKANTI, S., RATHA, N., AND SENIOR, A. *Guide to Biometrics*. Springer, 2003.
- [9] BRIK, V., BANERJEE, S., GRUTESER, M., AND OH, S. Wireless device identification with radiometric signatures. In *Proc. ACM MobiCom* (2008).
- [10] COSTEN, N., PARKER, D., AND CRAW, I. Effects of high-pass and low-pass spatial filtering on face identification. *Perception & Psychophysics* 58, 4 (1996), 602–612.
- [11] DANEV, B., AND ČAPKUN, S. Transient-based identification of wireless sensor nodes. In *Proc. ACM/IEEE IPSN* (2009).
- [12] DEVADAS, S., SUH, E., PARAL, S., SOWELL, R., ZIOLA, T., AND KHANDELWAL, V. Design and implementation of PUF-based “unclonable” RFID ICs for anti-counterfeiting and security applications. *Proc. IEEE Intl. Conf. on RFID* (2008), 58–64.
- [13] DIMITRIOU, T. A lightweight RFID protocol to protect against traceability and cloning attacks. In *Proc. ICST SecureComm* (2005).
- [14] DUC, D. N., PARK, J., LEE, H., AND KIM, K. Enhancing security of EPCglobal Gen-2 RFID tag against traceability and cloning. In *Proc. Symposium on Cryptography and Information Security* (2006).
- [15] ELLIS, K., AND SERINKEN, N. Characteristics of radio transmitter fingerprints. *Radio Science* 36 (2001), 585–597.
- [16] EPCGLOBAL. Architecture framework v. 1.2. standard, 2007. http://www.epcglobalinc.org/standards/architecture/architecture_1.2-framework-20070910.pdf.
- [17] FELDHOFFER, M., DOMINIKUS, S., AND WOLKERSTORFER, J. Strong authentication for RFID systems using the AES algorithm. In *Workshop on Cryptographic Hardware and Embedded Systems* (2004), M. Joye and J.-J. Quisquater, Eds., vol. 3156 of *LNCS*, pp. 357–370.
- [18] GRUNWALD, L. Cloning ePassports without active authentication. In *BlackHat* (2006).
- [19] HALAMKA, J., JUELS, A., STUBBLEFIELD, A., AND WESTHUES, J. The security implications of VeriChip™ cloning. Manuscript in submission, 2006.
- [20] HALL, J., BARBEAU, M., AND KRANAKIS, E. Enhancing intrusion detection in wireless networks using radio frequency fingerprinting. In *Proc. CIIT* (2004).
- [21] HALL, J., BARBEAU, M., AND KRANAKIS, E. Radio frequency fingerprinting for intrusion detection in wireless networks. *Submission to IEEE TDSC (Electronic Manuscript)* (2005).
- [22] HALL, J., BARBEAU, M., AND KRANAKIS, E. Detecting rogue devices in bluetooth networks using radio frequency fingerprinting. In *Proc. CCN* (2006).
- [23] HIPPENSTIEL, R., AND PAYAL, Y. Wavelet based transmitter identification. In *Proc. ISSPA* (1996).
- [24] JAIN, A., PRABHAKAR, S., AND CHEN, S. Combining multiple matchers for a high security fingerprint verification system. In *Pattern Recognition Letters* (1999).
- [25] JUELS, A. Minimalist cryptography for low-cost RFID tags. In *Intl. Conf. on Security in Communication Networks* (2004), C. Blundo and S. Cimato, Eds., vol. 3352 of *LNCS*, pp. 149–164.
- [26] JUELS, A. Strengthening EPC tags against cloning. Manuscript, 2005.
- [27] JUELS, A. Rfid security and privacy: A research survey. *IEEE Journal on Selected Areas in Communications* 24, 2 (2006).
- [28] JUELS, A., PAPPU, R., AND PARNO, B. Unidirectional key distribution across time and space with applications to RFID security. In *Proc. 17th USENIX Security Symposium* (2008), pp. 75–90.
- [29] JUELS, A., RIVEST, R., AND SZYDLO, M. The blocker tag: Selective blocking of RFID tags for consumer privacy. In *Proc. ACM CCS* (2003), pp. 103–111.
- [30] KAPLAN, D., AND STANHOPE, D. Waveform collection for use in wireless telephone identification, 1999.
- [31] KERSCHBAUM, F., AND SORNIOTTI, A. RFID-based supply chain partner authentication and key agreement. In *Proc. ACM WiSec* (2009).

- [32] KITTLER, J., HATEF, M., DUIN, R., AND MATAS, J. On combining classifiers. *IEEE Trans. on Pattern Analysis and Machine Intelligence* 20, 3 (1998).
- [33] LAURIE, A. Reading ePassports with predictable document numbers. In *news report* (2006).
- [34] M, W. Cloning ePassports with active authentication enabled. In *What The Hack* (2005).
- [35] MANLY, B. *Multivariate Statistical Methods: A Primer*, 3rd ed. Chapman & Hall, 2004.
- [36] MARPLE, S. Computing the discrete-time analytic signal via FFT. *IEEE Trans. on Signal Processing* 47, 9 (1999).
- [37] MITRA, M. Privacy for RFID systems to prevent tracking and cloning. *Intl. Journal of Computer Science and Network Security* 8, 1 (2008), 1–5.
- [38] OPPENHEIM, A., SCHAFER, R., AND BUCK, J. *Discrete-Time Signal Processing*, 2nd ed. Prentice-Hall Signal Processing Series, 1998.
- [39] O.URETEN, AND N.SERINKEN. Wireless security through RF fingerprinting. *Canadian J. Elect. Comput. Eng.* 32, 1 (Winter 2007).
- [40] RASSMUSSEN, K., AND CAPKUN, S. Implications of radio fingerprinting on the security of sensor networks. In *Proc. SecureComm* (2007).
- [41] RICHTER, H., MOSTOWSKI, W., AND POLL, E. Fingerprinting passports. In *NLUUG Spring Conference on Security* (2008).
- [42] ROSS, A., AND JAIN, A. Multimodal biometrics: An overview. In *Proc. EUSIPCO* (2004).
- [43] TEK BAS, O., URETEN, O., AND SERINKEN, N. Improvement of transmitter identification system for low SNR transients. In *Electronic Letters* (2004).
- [44] TOONSTRA, J., AND KISNER, W. Transient analysis and genetic algorithms for classification. In *Proc. IEEE WESCANEX* (1995).
- [45] TOONSTRA, J., AND KISNER, W. A radio transmitter fingerprinting system ODO-1. In *Canadian Conf. on Elect. and Comp. Engineering* (1996).
- [46] VAJDA, I., AND BUTTYÁN, L. Lightweight authentication protocols for low-cost RFID tags. In *Proc. 2nd Workshop on Security in Ubiquitous Computing – Ubicomp* (2003).
- [47] VANBEEK, J. ePassports reloaded. In *BlackHat* (2008).
- [48] WANG, B., OMATU, S., AND ABE, T. Identification of the defective transmission devices using the wavelet transform. *IEEE PAMI* 27, 6 (2005), 696–710.

CCCP: Secure Remote Storage for Computational RFIDs

Mastooreh Salajegheh¹ Shane Clark¹ Benjamin Ransford¹ Kevin Fu¹ Ari Juels²

¹*Department of Computer Science, University of Massachusetts Amherst*

²*RSA Laboratories, The Security Division of EMC*

{negin, ssclark, ransford, kevinfu}@cs.umass.edu, ajuels@rsa.com

Abstract

Passive RFID tags harvest their operating energy from an interrogating reader, but constant energy shortfalls severely limit their computational and storage capabilities. We propose *Cryptographic Computational Continuation Passing* (CCCP), a mechanism that amplifies programmable passive RFID tags' capabilities by exploiting an often overlooked, plentiful resource: low-power radio communication. While radio communication is more energy intensive than flash memory writes in many embedded devices, we show that the reverse is true for passive RFID tags. A tag can use CCCP to checkpoint its computational state to an untrusted reader using less energy than an equivalent flash write, thereby allowing it to devote a greater share of its energy to computation.

Security is the major challenge in such remote checkpointing. Using scant and fleeting energy, a tag must enforce confidentiality, authenticity, integrity, and data freshness while communicating with potentially untrustworthy infrastructure. Our contribution synthesizes well-known cryptographic and low-power techniques with a novel flash memory storage strategy, resulting in a secure remote storage facility for an emerging class of devices.

Our evaluation of CCCP consists of energy measurements of a prototype implementation on the batteryless, MSP430-based WISP platform. Our experiments show that—despite cryptographic overhead—remote checkpointing consumes less energy than checkpointing to flash for data sizes above roughly 64 bytes. CCCP enables secure and flexible remote storage that would otherwise outstrip batteryless RFID tags' resources.

1 Introduction

Research involving low-energy computing systems has long treated radio as an energy-hungry resource to be used sparingly. Our work uncovers a key resource in which programmable passive RFID tags differ

from higher-powered wireless embedded devices such as motes: *radio communication consumes less energy than persistent local storage*. We exploit radio as a resource to amplify the storage capabilities of an emerging class of batteryless, programmable devices called *computational RFIDs* (CRFIDs) [7, 26, 27].

The main idea of this paper is that a CRFID can securely use radio communication as a less energy-intensive alternative to local, flash-based storage. The smaller energy requirements of radio allow the CRFID either to devote more energy to computation or to accomplish the same tasks using less energy, which may translate into a longer operating range. We use established cryptographic mechanisms to protect against untrustworthy RFID readers that could attempt to violate the confidentiality, authenticity, integrity, and freshness of the data on a CRFID. However, the cryptographic overhead threatens to eliminate the energy advantage of remote storage. Thus, the main challenge is to design an energy-saving remote storage system that provides security under the constraints of passive RFID systems.

This paper uses computational state checkpointing as an example of an application that benefits from our techniques. *Cryptographic Computational Continuation Passing* (CCCP) enables CRFIDs to perform sophisticated computations despite limited energy and continual interruptions of power that lead to complete loss of the contents of RAM. CCCP extends the Mementos architecture [26] for execution checkpointing by securely storing a CRFID's computational state on the untrusted RFID reader infrastructure that powers the CRFID, thereby making program execution on CRFIDs robust against loss of power. The design of CCCP is motivated by (1) a desire to minimize the amount of energy devoted to flash memory writes and (2) the observation that a CRFID's backscatter transmission is surprisingly efficient compared to alternatives such as active radio (like that found in motes) or flash memory writes.

Our contribution in this paper is the synthesis of sev-

eral existing ideas with techniques that are specifically applicable to computational RFIDs:

- We describe the design and implementation of CCCP, a *secure remote storage protocol that suits the characteristics and constraints of CRFIDs*, and we show how this protocol can be used in the contexts of execution checkpointing and external data storage on an untrusted RFID reader infrastructure (Sections 3, 4).
- Motivated by a desire to save energy when storing CCCP's numeric counters to nonvolatile memory, we introduce *hole punching* (Section 3.4.4), a unary encoding technique that allows a counter stored in flash memory to be updated economically, minimizing energy- and time-intensive flash erase operations. For a CRFID, less frequent flash erasure means more energy available for computation.

Since CCCP involves communication with a potentially untrustworthy RFID reader, it must ensure the integrity, confidentiality, and data freshness of checkpointed messages. For message integrity, CCCP employs UMAC [4], a Message Authentication Code (MAC) scheme based on universal hash functions (UHF) that involves the application of a cryptographically secure pseudorandom pad. Remotely stored messages in CCCP are encrypted for confidentiality using a simple stream cipher. CCCP's frequent use of key material motivates the use of opportunistic precomputation: when a CRFID is receiving abundant energy, CCCP generates and stores keystream bits in flash memory for later consumption. CCCP maintains a small amount of its own state in local nonvolatile memory, including a counter that must be updated during checkpoint operations when energy may be low. To minimize the energy required to update the counter, CCCP employs hole punching.

Conventional passive RFID tags perform rudimentary computation, often in extremely tight real-time constraints using nonprogrammable finite state machines [1], but CRFIDs offer true general-purpose computational capabilities, broadening the range of their possible applications (Section 6). Although CRFIDs offer more flexibility, they present challenging resource constraints. While sensor motes, which rely on batteries for power, often have an active lifetime measured in weeks or months, a CRFID may be able to compute for less than a second given a burst of energy, and may receive such bursts in quick succession—putting CRFIDs in an entirely different class with regard to energy constraints. Moreover, although CRFIDs have a small amount of flash memory available as nonvolatile storage, writing to this flash memory is energy intensive (Section 2).

Because CRFIDs are new and prototypes are not yet widely available for use in the laboratory, there is lit-

tle previous work describing their applications or limitations; Section 7 summarizes relevant work that has appeared to date. CCCP extends a recent execution checkpointing system called Mementos [26] by adding remote, rather than local flash-based, storage capabilities to CRFIDs. While systems such as Mementos investigate how to effectively store checkpoints locally in trusted flash memory to achieve computational progress on CRFIDs despite power interruptions, CCCP focuses on using external, *untrusted* resources to increase tag storage capacity in a secure and energy-efficient manner.

2 Computational RFIDs: Background, Observations, Challenges

Consistent with the usage of RFID terminology, the term *Computational RFID* (CRFID) has two meanings: the *model* under which passively powered computers operate in concert with an RFID reader infrastructure, and the passively powered computers themselves. CRFIDs represent a class of programmable, batteryless computers [7, 26, 27]. The small size and low maintenance requirements of CRFIDs make them especially appealing for adding computational capabilities to contexts in which placing or maintaining a conventional computer would be infeasible or impossible. However, CRFID systems require that nearby, actively powered RFID readers provide energy whenever computation is to occur, a requirement that may not suit all applications.

The components of a CRFID are: a low-power microcontroller; onboard RAM; flash memory (on or off the microcontroller); energy harvesting circuitry tuned to a certain frequency (e.g., 913 MHz for EPC Gen 2 RFID); an antenna; a transistor between the antenna and the microcontroller to modulate the antenna's impedance; a capacitor for storage of harvested energy; one or more analog-to-digital converters; and optional sensors for physical phenomena such as acceleration, heat, or light. The first working example of a CRFID is the Wireless Identification and Sensing Platform, or WISP [29], a prototype device slightly smaller than a postage stamp (discounting its inches-long antenna). The WISP is built around an off-the-shelf TI MSP430 microcontroller.

Like passive RFID tags but unlike sensor motes, CRFIDs are powered solely by harvested RF energy and lack active radio components. Instead, such CRFIDs use *backscatter* communication: in the presence of incoming radio waves, a CRFID electrically modulates its antenna's impedance using a transistor, encoding binary information by varying the antenna's reflectivity. While the omission of active radio circuitry saves energy, it gives up the tag's autonomy; a CRFID can send and receive information only at the command of an RFID reader.

A CRFID's lack of autonomy is one of the factors that makes it difficult to protect.

2.1 Frequent Power Loss on Tags, but Plentiful External Resources

Several key observations motivate the development of secure remote storage for computational RFIDs.

Frequent loss of power may interrupt computation. The CRFID model posits computing devices that are primarily powered by RF energy harvesting, a mechanism that is naturally finicky because of its dependence on physical conditions. Any change to a CRFID's physical situation—such as its position or the introduction of an occluding body—may affect its ability to harvest energy. Existing systems that use RF harvesting typically counteract the effect of physical conditions by placing stringent requirements on use. For example, an RFID transit card reader presented with a card may behave in an undefined way unless the card is within 1 cm for at least 300 ms, parameters designed to ensure that the card's computation finishes while it is still near the reader. CRFID applications may preclude such a strategy: programs on general-purpose CRFIDs may not offer convenient execution time horizons, and communication distances may not be easily controlled. Without any guarantees of energy availability, it may be unreasonable to mandate that programs running on CRFIDs complete within a single energy lifecycle. As an extension of the Mementos system [26], CCCP aims to address the problem of suspending and resuming computations to facilitate spreading work across multiple energy lifecycles.

Storing remotely may require less energy than storing locally. Some amount of onboard nonvolatile memory exists on a CRFID, so an obvious approach to suspension and resumption is simply to use this local memory for state storage. However, to implement nonvolatile storage, current microcontrollers use flash memory, which imports several undesirable properties. While reading from flash consumes energy comparable to reading from volatile RAM, the other two flash operations—writing and erasing—require orders of magnitude more energy per datum (Table 3). Our measurements of a CRFID prototype reveal that the energy consumption of storing a datum locally in flash can in fact exceed the energy consumption of transmitting the same datum via backscatter communication.

To illustrate the difference between flash and radio storage on a CRFID and to show how the relationship is different on a sensor mote, we offer Figure 1. The figure helps explain why designers of mote-based systems choose to minimize radio communication; similarly, it justifies our exploration of radio-based storage as an alternative to flash-based storage on CRFIDs.

It should be noted that CCCP, although its primary data storage mechanism is the communication link between CRFIDs and readers, still requires *some* flash writes during storage operations: CCCP maintains a counter in flash to ensure that key material is not reused. However, because CCCP employs hole punching (Section 3.4.4) to maintain the counter, the amount of data written for counter updates is small compared to the amount of data that can be stored at once—small enough not to obviate the energy advantage of radio-based storage—and counter updates do not frequently necessitate erasures.

EPC Gen 2 RFID readers are typically not standalone devices. Rather, they are connected to networks or other systems (for, e.g., control or logging) that can offer computing resources such as storage. The benefit to CRFIDs that communicate with such a reader infrastructure is access to effectively limitless storage. Several kilobytes of onboard flash memory is minuscule compared to the potentially vast amount of storage available to networked RFID readers. While unlimited external storage is not obviously helpful for saving computational state—a CRFID cannot save or restore more state than it can hold locally—its usefulness as general-purpose long-term storage is analogous to the usefulness of networked storage for PCs.

RFID protocols allow arbitrary payloads. While the EPC Gen 2 protocol imposes constraints on the transmissions between RFID tags and RFID readers—for example, the maximum upstream data rate from tag to reader is 640 Kbps [13]—it also offers sufficient flexibility that CCCP can be implemented on top. In particular, the Gen 2 protocol permits a reader to issue a *Read* command to which a tag can respond with an arbitrary amount of data. Previous versions of the EPC RFID standard mandated a small response size that would have imposed severe communication overhead on large upstream transmissions.

CRFID is not married to EPC Gen 2 as an underlying protocol, but the existence of a widespread RFID reader infrastructure and the availability of commodity reader hardware makes for easy prototyping.

2.2 Challenges Due to Energy Scarcity

Several energy-related considerations limit the resources available for computation on CRFIDs, limiting the utility of CRFIDs as a general-purpose computing platform. Ransford et al. [26] discuss the difficulty of effectively utilizing a storage capacitor and enumerate the drawbacks of using capacitors for energy storage; Buettner et al. [7] discuss how energy limitations bear on the deployment of a CRFID-based system. Two key design features of CRFIDs pose energy challenges to a system like

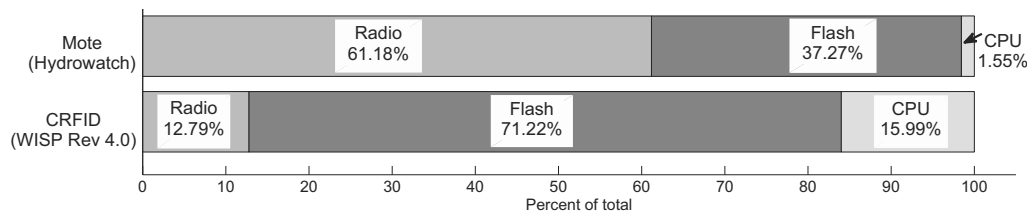


Figure 1: Per-component maximum power consumption of two embedded devices. Radio communication on the WISP requires less power than writes to flash memory. The relative magnitudes of the power requirements means that a sensor mote favors shifting storage workloads to local flash memory instead of remote storage via radio, while a computational RFID favors radio over flash. The numbers for the mote are calculated based on the current consumption numbers given by Fonseca et al. [15]. For the CRFID, we measured three operations (radio transmit, flash write, and register-to-register move) for a 128-byte payload.

CCCP: first, the voltage and current requirements of flash memory constrain the design of flash-bearing CRFIDs and limit the portion of a CRFID’s energy lifecycle that is usable for computation. Second, a CRFID’s reliance on energy harvesting and backscatter communication means that a CRFID cannot compute or communicate without reader contact.

Flash memory limitations. Microcontrollers that incorporate flash typically have separate threshold voltages: one threshold for computation, and a higher threshold for flash writes and erases. Because of this difference, flash writes cannot be executed at arbitrary times during computation on a CRFID; they require sufficient voltage on the storage capacitor. Without a constant supply of energy, capacitor voltage declines with time and computation, so waiting until the end of a computation to record its output to nonvolatile memory may be risky.

The size of a CRFID’s storage capacitor imposes another basic limitation. Flash writes, which owe their durability to a process that effects significant physical changes, require more current and time (and therefore energy) than much simpler RAM or register writes. Per-datum measurements show that, on a WISP’s microcontroller, writing to flash consumes roughly 400 times as much energy as writing to a register [26]. Such outflow from the storage capacitor can dramatically shorten the device’s energy lifecycles.

Non-autonomous operation. Backscatter communication involves modulating an antenna’s impedance to reflect radio waves—an operation that, for the sender, involves merely toggling a transistor to transmit binary data. Such communication cannot occur without a signal to reflect; CRFIDs, like other passive RFIDs, are therefore constrained to communicate only when a reader within range is transmitting. Computation may occur during times of radio silence, but only if sufficient energy remains in the CRFID’s storage capacitor. Unlike battery-powered platforms that can operate autonomously between beacon messages from other entities, a CRFID may completely lose power between in-

teractions with an RFID reader. Our experience shows that, lacking a source of harvestable energy, the storage capacitor on a WISP (Revision 4.0) can support roughly one second of steady computation before its voltage falls below the microcontroller’s operating threshold. Such limitations constrain the design space of applications that can run on CRFIDs. For example, without autonomy, an application cannot plan to perform an action at a specific time in the future.

Unsteady energy supply. A key challenge CRFIDs face is that their supply of energy can be unsteady and unpredictable, especially under changing physical conditions. RFID readers may not broadcast continuously or even at regular intervals, and they do not promise any particular energy delivery schedule to tags. In our experiments, even within inches of an RFID reader that emitted RF energy at a steady known rate, the voltage on a CRFID’s storage capacitor did not appear qualitatively easy to predict despite the fixed conditions. A CRFID’s storage capacitor must buffer a potentially unsteady supply of RF energy without the ability to predict future energy availability.

3 Design of CCCP

CCCP’s primary design goal is to furnish computational RFIDs with a mechanism for secure outsourced storage that facilitates the suspension and resumption of programs. This section describes how CCCP is designed to meet that goal and several others. Refer to Section 4 for a discussion of CCCP’s implementation, and refer to Section 5 for an evaluation of CCCP’s design choices and security; in particular, Section 5.3.1 discusses the overhead imposed by cryptographic operations.

Given a chunk of serialized computational state on a CRFID, CCCP sends the state to the reader infrastructure for storage. (CCCP is designed to work independently of the state serialization method, and does not prescribe a specific method.) In a subsequent energy lifecycle, an RFID reader that establishes communication with the tag

Design goal	Approach
Computational progress	Communicating checkpoints via radio to untrusted RFID readers
Security: authentication, integrity	UHF-based MAC
Security: data freshness	Key non-reuse; counter stored by hole punching in nonvolatile memory
Security: confidentiality	Symmetric encryption with keystream precomputation

Table 1: CCCP’s design goals and techniques for accomplishing each of them.

sends back the state, CCCP performs appropriate checks, and the CRFID resumes computation where it left off. CCCP provides several operating modes that allow an application designer to increase security—by adding authentication alone, or authentication and encryption—at the cost of additional per-checkpoint energy consumption. Table 1 describes how CCCP meets each of the goals discussed in this section.

3.1 Design Goal: Computational Progress on CRFIDs

CCCP remotely checkpoints computational state to make long-running operations robust against power loss—i.e., to enable their *computational progress*. We define computational progress as change of computational state toward a goal (e.g., the completion of a loop). While CRFIDs are able to finish short computations in a small number of energy lifecycles (e.g., symmetric-key challenge-response protocols [9, 19]), the challenges described in Section 2 make it difficult for a CRFID to guarantee the computational progress of longer-running computations.

If a CRFID loses power before it completes a computation, all volatile state involved in the computation is lost and must be recomputed in the next cycle. If energy availability is similarly inadequate in subsequent cycles, the CRFID may never obtain enough energy to finish its computation or even to checkpoint its state to flash memory. We refer to such vexatious computations as *Sisyphean tasks*. (Sisyphus was condemned to roll a large stone up a hill, but was doomed to drop the stone and repeat hopelessly forever [20].) A major goal of CCCP is to prevent tasks from becoming Sisyphean by shifting energy use away from flash operations and toward less energy-intensive radio communication.

3.2 Checkpointing Strategies: Local vs. Remote

We consider two strategies for the nonvolatile storage of serialized checkpointed state. The first, writing the state to flash memory, involves finding an appropriately sized region of erased flash memory or creating one via erase operations. The second strategy, using CCCP, requires

a CRFID to perform zero or more cryptographic operations (depending on the operating mode) and then transmit the result via backscatter communication.

The obvious advantage of flash memory is that its proximity to the CRFID makes it readily accessible. On-chip flash has the further advantage that it may be inaccessible to an attacker. However, the operating requirements of flash are onerous in many situations. With unlimited energy, a CRFID could use flash freely and avoid the complexity of a radio protocol such as CCCP. Unfortunately, energy is limited in ways described elsewhere in this paper, and several disadvantages of flash memory diminish its appeal as a store for checkpointed state. The most obvious disadvantage is an imbalance between the requirements for reading and writing. Write and erase operations require more time and energy per bit than reading (Table 3); additionally, the minimum voltage and current requirements are higher. For example, in the case of the MSP430F2274, read operations are supported at the microcontroller’s minimum operating voltage of 1.8 V, but write and erase operations require 2.2 V. Finally, flash memory (both NOR and NAND types) generally imposes the requirement that memory segments be erased before they are written: if a bit acquires a zero value, the entire segment that contains it must be erased for that bit to return to its default value of 1. Aside from burdening the application programmer with inconvenience, erase-before-write semantics complicate considerations of energy requirements. These disadvantages are minor afflictions for higher-powered systems, but they pose serious threats to the utility of flash memory on CRFIDs.

Backscatter transmission, since it involves modulating only a single transistor to encode data, requires significantly less energy than transmission via active radio. In fact, our measurements (Figure 4) show that backscatter transmission of an authenticated, encrypted state checkpoint (plus a small amount of bookkeeping in flash) can require less energy than exclusively writing to flash memory, even after including the energy cost of encrypting and hashing the checkpointed state. Because of its consistent behavior throughout the microcontroller’s operating voltage range, backscatter transmission is an especially attractive option when the CRFID receives radio contact frequently but cannot harvest energy efficiently,

in which case writing to flash may be infeasible because of insufficient energy in the storage capacitor. These circumstances may occur far from the reader, or in the presence of radio occlusions, or when a computation uses energy quickly as soon as the CRFID wakes up.

Despite its advantages over flash storage and active radio, CCCP's reliance on backscatter transmission has drawbacks. Bitrate limitations in the EPC Gen 2 protocol cause CCCP's transmissions to require up to twice as much time per datum as flash storage on some workloads. The best choice of storage strategy depends on an application's ability to tolerate delay and the necessity of saving energy.

3.3 Threat Model

We define CCCP's threat model as a superset of the attacks that typically threaten RFID systems [21]. The most obvious way an attacker can disrupt the operation of a CRFID is to starve it of energy by jamming, interrupting, or simply never providing RF energy for the CRFID to harvest. Because they depend entirely on harvestable energy, CRFIDs cannot defend against such denial-of-service (DoS) attacks, so we consider these attacks as a problem to be dealt with at a higher system level. We instead focus on two types of attacks that a CRFID can use its resources to address: (1) active and passive radio attacks and (2) attacks by an untrusted storage facility.

An adversary may attempt to:

- Eavesdrop on radio communication in both directions between a CRFID and reader.
- Masquerade as a legitimate RFID reader in order to collect checkpointed state from CRFIDs. Because CRFIDs do not trust reader infrastructure, such an attack should allow an attacker to collect only ciphertext.
- Masquerade as a legitimate RFID reader in order to send corrupted data or old data (e.g., a previous computational state) to the CRFID. Such invalid data should not trick the CRFID into executing arbitrary or inappropriate code.
- Masquerade as a specific legitimate CRFID in order to retrieve that CRFID's stored state from the reader. This state should be useless without access to the keystream material that encrypted it—keystream material that is stored in the legitimate owner's nonvolatile memory and never transmitted.

We additionally assume that an adversary cannot physically inspect the contents of a CRFID's memory.

3.4 Secure Storage in CCCP

Because computational RFIDs depend on RFID readers for energy—if a CRFID is awake, there is probably a reader nearby—readers are a natural choice for storing information. But a reader trusted to provide energy should not necessarily be trusted with sensitive information such as checkpointed state.

CCCP involves communication with untrusted reader infrastructure, so we establish several security goals:

- **Authenticity:** a CRFID that stores information on external infrastructure will eventually attempt to retrieve that information, and the authenticity of that information must be cryptographically guaranteed. Under CCCP, the only party that ever needs to verify the authenticity of a CRFID's stored information is the CRFID itself.
- **Integrity:** an untrusted reader may attempt to impede a CRFID's computational progress by providing data from which the CRFID cannot resume computation (e.g., random junk). While CCCP cannot prevent a denial of service attack in which a reader provides only junk, it guarantees that CRFIDs will compute only on data they recognize.
- **Data freshness:** just as a reader can provide corrupted data instead of usable data, it can replay old state in an attempt to hinder the computational progress of a computation. Under CCCP, a CRFID recognizes and rejects old state.
- **Confidentiality:** in certain applications, the leaking of intermediate computational state might be a critical security flaw. For other applications, confidentiality may not be necessary.

3.4.1 Keystream Precomputation

Because CCCP's threat model assumes a powerful adversary that can intercept all transmissions, CCCP never reuses keystream material when encrypting data or computing MACs. We use CCCP's refreshable pool of pseudorandom bits (a circular buffer in the CRFID's nonvolatile memory) as a cryptographic keystream to provide confidentiality and authentication.

CCCP stores keystream material on the CRFID because we assume that the CRFID trusts only itself; a CRFID cannot extract trustworthy keystream material from a reader it does not trust, nor from any observable external phenomenon (which, in our threat model, an attacker would be able to observe equally well). Because a CRFID can reserve only finite storage for storage of keystream material, the material must be periodically refreshed. CCCP opportunistically refreshes the keystream material with pseudorandom bits, following Algorithm 3.

To provide unique keystream bits to cryptographic operations (encryption and MAC), CCCP uses an existing implementation [9] of the RC5 block cipher [28] in counter mode to generate pseudorandom bits and store them to flash. The choice of a block cipher in counter mode means that the resulting MAC and ciphertext are secure against a computationally bounded adversary [6]. A stream cipher would work equally well in principle, but in implementing CCCP, we found that those under consideration required a large amount of internal read-write state. For example, the stream cipher ARC4 requires at least 256 bytes of RAM [30], whereas RC5 requires only an 8-byte counter. The RC5 key schedule is preloaded into flash memory the first time the device is programmed, and the keystream materials are generated during periods of excess energy (or *power seasons*; see § 3.5). One such period of excess energy is the CRFID’s initial programming, at which time the entire keystream buffer is filled with keystream bits. To avoid reusing keystream bits, CCCP maintains several variables in nonvolatile memory. Table 2 summarizes the variables CCCP stores in nonvolatile memory.

Variable	Description
<i>chkpt_counter</i>	Counter representing the number of checkpoints completed; used to calculate the location of the first unused keystream material; updated each time keystream material is consumed; unary representation
<i>kstr_end</i>	Pointer to the end of the last chunk of unused keystream bits in keystream memory; updated during key refreshment
<i>rc5counter</i>	Incrementing counter used as an input to RC5 while filling keystream memory with pseudorandom data; updated during key refreshment

Table 2: Variables CCCP stores in nonvolatile memory.

3.4.2 UHF-based MAC for Authentication and Integrity

CCCP uses a MAC scheme based on universal hash functions (UHF) [8] to provide authentication and integrity. CCCP constructs the MAC by first hashing the message and then XORing the 80-bit hash with a precomputed cryptographic keystream. Because of the resource constraints of CRFIDs, it is critical to use a scheme that consumes minimal energy, and according to recent literature [4, 14], UHF-based MACs are potentially an order of magnitude faster than MACs based on cryptographic

hash functions. We chose UMAC [4] as the MAC function after evaluating several alternatives. Our experiments on WISP (Revision 4.0) CRFIDs determined that UMAC takes on average 18.38 ms and requires 28.79 μ J of energy given a 64-byte input.

3.4.3 Stream Cipher for Confidentiality

To provide confidentiality, a CRFID simply XORs its computational state with a precomputed cryptographic keystream. This encryption scheme is low-cost in terms of computation and energy, but it relies on using each keystream bit at most once. CCCP ensures that the encryption and MAC functions never reuse keystream bits by keeping track of the beginning and end of fresh keystream material in flash memory. The keystream pool is represented as a circular buffer. The address of the first unused keystream material is derived from the value of *chkpt_counter* (Table 2), and the last unused keystream material ends just before the address pointed to by *kstr_end*.

If the application using CCCP demands confidentiality at all times, then if CCCP cannot satisfy a request for unused keystream bits, it pauses its work to generate more keystream bits. This behavior is inspired by that of the blocking random device in Linux [17].

3.4.4 Hole Punching for Counters Stored in Flash

To avoid reusing keystream material, CCCP maintains a counter (*chkpt_counter*) from which the address of the first unused keystream bits can be derived. The counter is stored in flash memory because it is used for state restoration after power loss. However, incrementing a counter stored in binary representation always requires changing a 0 bit into a 1 bit (Figure 2(a)). On segmented flash memories, changing a single bit to 1 requires the erasure (setting to 1) of the entire segment that contains it—at least 128 bytes on the MSP430F2274—before the new value can be written. An additional cost that varies among flash cells is that they wear out with repeated erasure and writing [18].

To avoid energy-intensive erasures and minimize the energy cost of writing counter updates, CCCP represents *chkpt_counter* in complemented unary instead of binary. CCCP interprets the value of such a counter as the number of 0 bits therein. Because 1 bits can be changed to 0 bits without erasure, incrementing a counter requires a relatively small write, with erasures necessary only if the unary counter must be extended into unerased memory. We call this technique *hole punching* after the visual effect of turning 1 bits into 0 bits. Since *chkpt_counter* is simply incremented at each remote checkpoint, updating the counter generally requires writing only a single

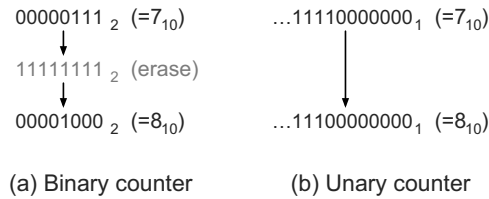


Figure 2: Illustration of hole punching. While incrementing a binary counter (a) in flash memory may require an energy-intensive erase operation, complemented unary representation (b), with the number of zeros, or “holes,” representing the counter value) allows for incrementing without erasure at a cost of space efficiency.

word. Table 3 illustrates the energy cost of erasing an entire segment and the energy cost of writing a single word.

Operation	Seg. erase	Write	Read	Write
Size (bytes)	128	128	128	2
Energy (μ J)	46.81	56.97	0.64	0.96

Table 3: Comparison of energy required for flash operations on an MSP430F2274. Hole punching often allows CCCP to use a single-word write (2 bytes on the MSP430) instead of a segment erase when incrementing a complemented unary counter.

To minimize the length of the unary *chkpt_counter*’s representation and to facilitate simple computation of offsets, CCCP assumes a fixed size for checkpointed state; in practice an application designer can choose an appropriate value for the fixed checkpoint size.

3.4.5 Extension for Long-Term Storage

Under CCCP, readers can act not only as outsourced storage for computational state, but also as long-term external storage. Because of their ultra-low-power microcontrollers, CRFIDs are likely to have only a small amount of flash available for data storage. Moreover, since flash operations are energy intensive, depending exclusively on flash memory as a storage medium is undesirable. CCCP could enable a CRFID to instead use the reader infrastructure as an external storage facility with effectively limitless space.

Long-term storage requires a different key management strategy than checkpointing data. With a temporary checkpointing system, the CRFID needs access only to the keystream material used to prepare the last checkpoint sent to a reader. However, in the case of long-term storage, the CRFID may require access to all of the data it has ever stored on the reader and therefore must remember all of the cryptographic keys from those stores. To avoid this unrealistic requirement, a potential extension

to CCCP allows the CRFID to generate keys on demand when long-term storage is required.

There are two operations that CCCP can provide to a CRFID application for this purpose:

- To satisfy a *STORE(data)* request, CCCP provides a keystream generator in the form of a block cipher in counter mode; this requires a monotonically increasing counter in addition to CCCP’s *chkpt_counter*. CCCP XORs the given data with the generated keystream and then constructs a MAC, then sends the ciphertext and MAC to the reader for storage. CCCP then sends the counter value back to the application.
- To satisfy a *RETRIEVE(index)* request, CCCP asks the RFID reader for the data at the given index. CCCP then generates the same keystream it used to encrypt the data by passing the index to the block cipher. Finally, CCCP verifies the MAC provided by the reader and returns the decrypted data to the application.

3.5 Power Seasons

If a CRFID could predict future energy availability, then it would be able to schedule its generation of keystream bits and ensure that it never exhausted its supply of pseudorandomness during normal operation. However, because CRFIDs lack autonomy and cannot depend on RFID reader infrastructure to provide a steady energy supply, we roughly classify the energy availability scenarios a CRFID faces into two *seasons*. We assume that the general case is a *winter* season, in which a CRFID cannot consistently harvest enough energy to perform all of its tasks. During winter, the CRFID must focus on minimizing checkpoints and wasted energy. The other season is *summer*, during which harvested energy is plentiful and the CRFID can afford to perform energy-intensive operations such as precomputation and storage of keystream material for later use.

CCCP can identify a summer season if one of two conditions is true. First, the CRFID may find itself awake with no computations left to complete, for example after it has finished a sensor reading. Second, the CRFID may find itself communicating with a reader that does not understand CCCP and simply provides harvestable energy.

4 Implementation

The components of CCCP span two environments: CRFIDs and RFID readers. On a CRFID, CCCP accepts data from an application and uses the CRFID’s backscatter mechanism to ship the data to a reader. The reader

(which we consider as an RFID reader plus a controlling computer) is programmed to participate in the CCCP protocol and return computational state where necessary. This section describes the CRFID-side components, the reader-side components, and the protocol that ties them together.

The CRFID side of CCCP is implemented in the C programming language on WISP (Revision 4.0) prototypes. At its core are three primary routines, which we present in pseudocode: CHECKPOINT (Algorithm 1), RESUME (Algorithm 2), and KEY-REFRESH (Algorithm 3). CHECKPOINT and RESTORE refer to a counter called *chkpt_counter* from which CCCP derives the address of the first unused keystream material. For routines that require radio communication, we borrow radio code from Intel's WISP firmware version 1.4. Note that, since a CRFID cannot assume that a reader is listening at an arbitrary time, the TRANSMIT subroutine waits for an interrupt indicating that the CRFID has received a go-ahead message from the reader.

The RFID reader side of CCCP consists only of code to drive the reader appropriately for communication events. Because of the Gen 2 protocol's complexity, we have not completely implemented the reader side of the CCCP protocol. Rather than write a large amount of code for the reader, we chose to use simple control programs for the reader and inspect the exchanged messages manually, a strategy that allowed us to concentrate on the more resource-constrained CRFID side of the system while avoiding porting applications from one proprietary reader to another. (The WISP [Revision 4.0] is nominally compatible with only the Alien ALR-9800 and Impinj Speedway readers; we chose to use a desktop PC to program these readers for the sake of simplicity and portability.) A full implementation of the reader side would properly parse each message received from the CRFID and manage storage for checkpointed state.

4.1 Communication Protocol

The CRFID model places a number of restrictions on communication. The only communication hardware on a CRFID is a backscatter circuit involving an antenna and a modulating transistor; an active radio would require significantly more energy. Since backscatter simply reflects an incoming carrier signal, a prerequisite for communication is that the reader emits an appropriate carrier signal. In our experiments, we used two different EPC Gen 2-compatible RFID readers that are readily available as off-the-shelf products; we used no nonstandard reader hardware or antennas.

CCCP's communication protocol is based on primitives provided by the EPC Gen 2 RFID protocol (the

RFID protocol the WISP understands). Specifically, CCCP makes use of three EPC Gen 2 commands:

- A reader issues a *Query* command to a specific tag (in our case, a CRFID). The *Query* command comprises a 4-tuple: $\langle action, membank, pointer, length \rangle$. While a conventional RFID tag may require reasonable values for all four tuple members, a CRFID need examine only the fourth member to learn the maximum reply length the reader will accept. The reader can use the other three fields to encode meta-information such as whether the reader wants to offer checkpointed state to the CRFID.
- A reader issues a *Read* command to a specific tag to request an arbitrary amount of data from an RFID tag's memory. A CRFID can respond to a coordinated *Read* command with a chunk of checkpointed state.
- A reader issues a *Write* command to send data for storage in a specific tag's memory. Because RFID tags tend to have fewer resources even than CRFIDs, *Write* commands transmit only a small amount (16 bits) of data. A CRFID can request a series of *Write* commands from the reader to retrieve checkpointed state, then reassemble the results in memory and restore its state from the checkpoint.

Figure 3 gives an overview of CCCP's message types and their ordering. CCCP does not require protocol changes to the EPC Gen 2 standard, but it requires that an RFID reader be controlled by an application that understands CCCP. While a proprietary radio protocol for CCCP could be more efficient than one built atop an existing RFID protocol, a goal of CCCP—inherited from the design goals of the WISP CRFID—is to maintain compatibility with existing RFID readers.

5 System Evaluation

This section justifies our design choices and offers evidence for our previous claims. We evaluate the security properties of four distinct checkpointing strategies—three based on CCCP's radio transmission and one on local flash storage—and describe how CCCP provides data integrity with or without confidentiality. We describe our experimental setup and methods, then provide empirical evidence that CCCP's radio-based checkpointing requires less energy per checkpoint than a flash-based strategy. Finally, we characterize the overhead incurred by CCCP's cryptographic operations in terms of both energy and the keystream material that they consume.

Algorithm 1 The CHECKPOINT routine encrypts, MACs, and transmits a fixed-size ($STATE_SIZE$, selected by the application designer) chunk of computational state. $\langle A, B \rangle$ means the concatenation of A and B with a delimiter in between. 80 bits is the fixed output size of NH, the hash function used by UMAC. For arithmetic simplicity, this pseudocode treats the *keystream* pool as an infinite array.

CHECKPOINT($state, keystream, chkpt_counter$)

```

1   $\triangleright$  Compute the (constant) amount of keystream material that will be used in this invocation
2   $chkpt\_size = STATE\_SIZE + LENGTH(\langle state, chkpt\_counter \rangle) + 80$  bits
3
4   $k \leftarrow chkpt\_counter \times chkpt\_size$   $\triangleright$   $keystream[k]$  holds unused keystream material
5   $chkpt\_counter \leftarrow chkpt\_counter + 1$   $\triangleright$  Update  $chkpt\_counter$  in nonvolatile memory
6
7   $C \leftarrow state \oplus keystream[k \dots k + STATE\_SIZE - 1]$   $\triangleright$  Encrypt  $state$  by XORing with keystream material
8   $k \leftarrow k + STATE\_SIZE$   $\triangleright$  ... and advance  $k$ 
9
10  $H \leftarrow NH(\langle C, k \rangle, keystream[k \dots k + LENGTH(\langle C, k \rangle) - 1])$   $\triangleright$  Hash the encrypted state
11  $k \leftarrow k + LENGTH(\langle C, k \rangle)$   $\triangleright$  ... and advance  $k$ 
12
13  $M \leftarrow H \oplus keystream[k \dots k + LENGTH(H) - 1]$   $\triangleright$  Construct an 80-bit MAC
14
15 TRANSMIT( $C, M$ )  $\triangleright$  Note: TRANSMIT blocks until a reader is detected
```

5.1 Security Semantics

CCCP trades the physical security of local storage for the energy savings of remote storage, but its use of radio communications introduces different security properties. We consider CCCP's four operating modes in increasing order of cryptographic complexity. Note that the algorithm listings (Algorithms 1–3) describe the most computationally intensive operating mode; the other modes involve subsets of its operations.

- Under CCCP's threat model, storing checkpointed state only in local flash memory is the most secure option, since it involves no radio transmission at all. However, for reasons detailed elsewhere in this paper, writing to flash memory is not always possible or desirable. We call the flash-only approach *Mementos* after the system [26] that inspired CCCP.
- In a mode called *CCCP/NoSec*, a CRFID sends computational state in plaintext. Under CCCP's threat model, CCCP/NoSec allows an attacker to intercept computational state and trivially recover the information it contains.
- In a mode called *CCCP/Auth*, the CRFID computes a message authentication code (MAC), attaches it to plaintext computational state, and transmits both. To trick a CRFID into accepting illegitimate state, an attacker must craft a message that incorporates a MAC that the CRFID can verify. However, since CCCP's MAC routine incorporates

keystream material that is local to the CRFID, the attacker must guess the contents of a chunk of the CRFID's keystream memory, which requires brute force under our threat model.

- In a mode called *CCCP/AuthConf*, CCCP encrypts computational state, computes a MAC, and transmits both (Algorithm 1). As with CCCP/Auth, an attacker who wants to trick a CRFID into accepting illegitimate state must find a hash collision; however, part of her colliding input must be a valid *encrypted* computational state from which the CRFID would be able to resume. Since CCCP does not reuse keystream material, the attacker is limited to brute-force search to find such an encrypted state.

5.2 Experimental Setup & Methods

We used a consistent experimental setup to obtain timing and energy measurements for a prototype CRFID. We programmed a WISP with a task (e.g., a flash write) and set a GPIO pin to toggle immediately before and after the task. We then charged the WISP's capacitor to 4.5 V using a DC power supply, disconnected the power supply so that the storage capacitor was the only source of energy for the WISP, and observed the task's execution and storage capacitor's voltage on an oscilloscope. We delivered energy directly from a DC power supply when taking measurements because the alternative, providing an RF energy supply, results in unpredictable and unsteady

Algorithm 2 The RESUME routine receives an encrypted checkpoint C and a message authentication code M from a reader, then restores the computational state of the CRFID if the received data pass an authenticity test. $chkpt_counter$ is the value stored in nonvolatile memory at the beginning of CHECKPOINT (Algorithm 1). We assume that, since k and $chkpt_counter$ are both numbers, their in-memory representations have the same length. As in Algorithm 1, this pseudocode treats the *keystream* pool as an infinite array for arithmetic simplicity. $\langle A, B \rangle$ means the concatenation of A and B with a delimiter in between. 80 bits is the fixed output size of NH, the hash function used by UMAC.

RESUME($C, M, keystream, chkpt_counter$)

```

1  > Find the first unused keystream material, then backtrack to find the keystream material CHECKPOINT used to
    hash and MAC the ciphertext
2   $chkpt\_size = STATE\_SIZE + LENGTH(\langle C, chkpt\_counter \rangle) + 80 \text{ bits}$            > N.b.:  $STATE\_SIZE = LENGTH(C)$ 
3   $k \leftarrow chkpt\_counter \times chkpt\_size$ 
4   $k \leftarrow k - (LENGTH(\langle C, k \rangle) + 80 \text{ bits})$ 
5
6   $H \leftarrow NH(\langle C, k \rangle, keystream[k \dots k + LENGTH(\langle C, k \rangle) - 1])$            > Compute the ciphertext's hash
7   $k \leftarrow k + LENGTH(\langle C, k \rangle)$                                            > ... and advance  $k$  to point to the MAC
8
9  if  $M = H \oplus keystream[k \dots k + LENGTH(H) - 1]$                        > If the MAC is OK, then...
10     then  $k \leftarrow k - (LENGTH(C) + LENGTH(\langle C, k \rangle))$                  > backtrack further...
11          $state = C \oplus keystream[k \dots k + LENGTH(C) - 1]$            > and decrypt  $C$  to yield  $state$ 
12         RESTORE-STATE( $state$ )
13     else > Do nothing

```

charge accumulation, making it difficult to shut off the energy supply at a precise capacitor voltage.

After watching the GPIO pin signal the beginning and end of the task, we calculated the task's duration and the corresponding change in the storage capacitor's voltage. When an operation completed too quickly to observe clearly on the oscilloscope, we repeated it in an unrolled loop and divided our measurements by the number of repetitions. Finally, we calculated per-bit energy values by subtracting the baseline energy consumption of the WISP with its MSP430 microcontroller in the LPM3 low-power (sleep) mode. We subtract the WISP's baseline energy consumption in order to discount the effects of omnipresent consumers such as RAM and CPU clocks. For all measurements that we present, we give the average of five trials.

5.3 Performance

Figure 4 shows that, for data sizes greater than 16 bytes, a checkpoint operation under CCCP/NoSec requires less energy than a checkpoint to flash. Under CCCP/AuthConf, which adds encryption and MAC operations, a similar threshold exists between 64 and 128 bytes. Checkpointing via flash has an additional cost: if the checkpointing mechanism needs to overwrite existing data (e.g., old checkpoints) in flash memory, it must erase the corresponding flash segments and poten-

tially replace whatever data it did not overwrite. Even if a flash write does not necessitate an immediate erasure, it makes less space available in the flash memory and therefore increases the probability that a long-running application will eventually need to erase the data it wrote—that is, it incurs an *energy debt*. In the ideal case, an application can pay its energy debt easily if erasures happen to occur only when energy is abundant—i.e., in summer power seasons. However, since CCCP is designed to address scenarios in which energy availability fluctuates, we consider the case in which each write incurs an energy debt. Factoring in debt, we characterize the energy cost of a write of size $dsize$ as

$$\text{Cost}^*(\text{write}(dsize)) = \text{Cost}(\text{seg. erase}) \times \frac{dsize}{\text{Size}(\text{seg.})} + \text{Cost}(\text{write}(dsize)).$$

In practice, because some erasures will likely occur in summer power seasons and some in winter power seasons, the energy cost of a flash write of size $dsize$ falls between $\text{Cost}(\text{write}(dsize))$ (the ideal cost) and $\text{Cost}^*(\text{write}(dsize))$ (the worst-case cost), inclusive.

The energy measurements we present in this paper (e.g., in Figure 4) fail in some cases to strongly support the hypothesis that radio-based checkpointing is consistently less energy intensive than flash-based checkpointing. The imbalance is due to a missed opportunity for optimization on the WISP prototype. The transistor used

Algorithm 3 The KEY-REFRESH replaces used keystream material with new keystream material in nonvolatile memory. Unlike in CHECKPOINT and RESUME, this pseudocode treats the *keystream* pool as a fixed-size circular buffer. This allows us to treat keystream material between k and $kstr_end$ as unused, and the rest—between $kstr_end$ and k —as used. This pseudocode omits two subtleties for simplicity: first, the routine must not erase keystream material that is waiting to be used by RESUME. Second, because flash erasure affects entire segments at once, the ERASE-MEMORY-RANGE routine must sometimes restore data that should not have been erased.

KEY-REFRESH(*keystream*, $kstr_end$, $chkpt_counter$, $rc5counter$)

```

1  > Find the first unused keystream material in the circular keystream buffer
2   $chkpt\_size = STATE\_SIZE + (STATE\_SIZE + LENGTH(\langle null, chkpt\_counter \rangle)) + 80$  bits
3   $k \leftarrow chkpt\_counter \times chkpt\_size \pmod{LENGTH(keystream) / chkpt\_size}$ 
4
5  > Erase all used keystream memory, then write pseudorandom data to it
6  ERASE-MEMORY-RANGE(keystream[ $kstr\_end \dots k$ ])
7   $i \leftarrow kstr\_end$ 
8  while ( $i < k$ )
9      do  $rc5counter \leftarrow rc5counter + 1$                                 > Update counter in nonvolatile memory
10          $keystream[i] \leftarrow RC5(rc5counter - 1)$                 > Write keystream material into nonvolatile memory
11          $kstr\_end \leftarrow i + 1$                                 > Update  $kstr\_end$  in nonvolatile memory
12          $i \leftarrow i + 1$ 
```

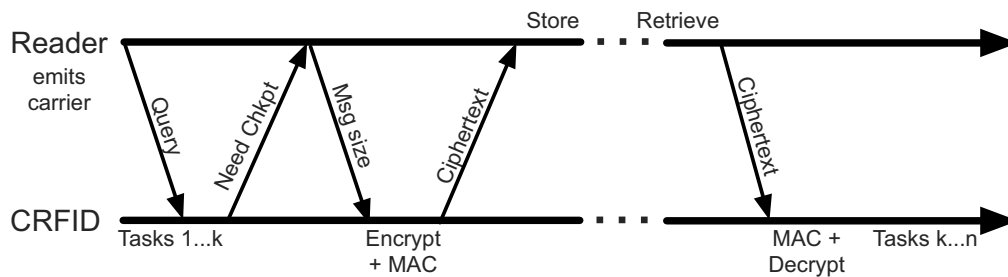


Figure 3: Application-level view of the CCCP protocol. The CRFID sends a request to checkpoint state while in the presence of a reader, and the reader specifies the maximum size of each message. The CRFID then prepares the checkpoint and transmits it in a series of appropriately sized messages. The reader stores the checkpoint data for later retrieval by the CRFID. All messages from the reader to the CRFID also supply power to the CRFID if the latter is within range.

for backscatter modulation on the WISP (Revision 4.0) draws $500 \mu\text{W}$ of power, far more than is typical of a comparable mechanism on a conventional RFID tag. Alien’s Higgs 3, a conventional RFID tag, draws only $15.8 \mu\text{W}$ of power [2] (total) during operation—an order of magnitude difference that supports an alternative design choice for future CRFIDs.

5.3.1 System Overhead

An application on a CRFID can balance energy consumption against security by choosing one of CCCP’s operating modes:

- CCCP/NoSec imposes the least overhead because it does not encrypt data or compute a MAC; it requires

no computation and consumes no keystream material. However, CCCP/NoSec imposes a time overhead to receive computational state from a reader at power-up and to transmit new state at checkpoint time.

- CCCP/Auth avoids encryption overhead (like CCCP/NoSec) but requires time, energy, and keystream bits to compute a MAC over the plaintext checkpoint. However, it requires no energy or keystream bits for encryption because it does not encrypt the plaintext checkpoint.
- CCCP/AuthConf offers the most security, since it adds confidentiality to CCCP/Auth, but the extra security comes at the expense of time, energy, and keystream bits. In this mode, CCCP encrypts the computational state before computing a MAC and

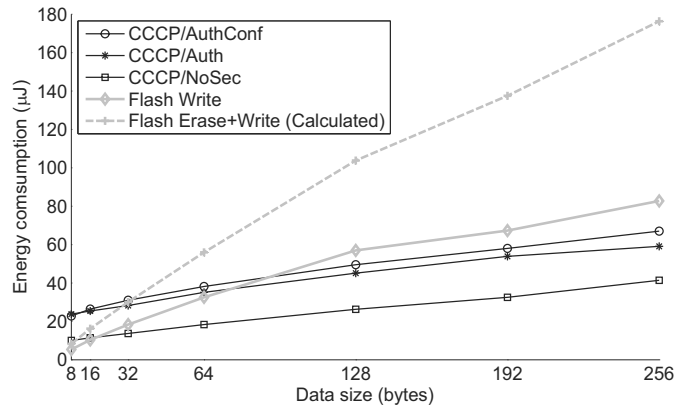


Figure 4: Energy consumption measurements from a WISP (Revision 4.0) prototype for all considered checkpointing strategies. Under our experimental method, we are unable to execute flash writes larger than 256 bytes on current hardware because larger data sizes exhaust the maximum amount of energy available in a single energy lifecycle. The average and maximum percent error of the measurements are 5.85% and 14.08% respectively.

transmitting both. It requires as much keystream material as the size of the state plus a constant amount for authentication.

6 Applications

The outsourced memory introduced by CCCP expands the design space for applications on a computational RFID. This section offers some illustrative example applications.

CRFIDs as low-maintenance sensors. Consider a *cold-chain monitoring* application for pharmaceutical supplies, in which a CRFID carries an attached temperature sensor and stores in flash memory a temperature reading each time it is scanned. To prevent exhaustion of its flash memory, the CRFID periodically computes aggregate statistics on, then discards, stored readings. Some statistical computations (e.g., computation of quartiles) require memory-intensive manipulation of the data set. If the flash memory on the CRFID considerably exceeds the size of RAM, computation of such statistics would require many writes to flash, an energy-intensive operation. An alternative is to use outsourced memory for the computation. (In the case of cold-chain monitoring, maintaining privacy of harvested data with respect to the reader may be unessential, but the *integrity* of the statistical computation is important.)

RFID sensor networks. Recent work [7] describes *RFID sensor networks* (RSNs) that combine RFID reader infrastructure with sensor-equipped computational RFIDs. RSNs do not simply replace traditional sensor networks because of several limitations. First, they require an infrastructure of readers that provide power to sensor nodes. Second, they are constrained by the distances (several meters) at which CRFIDs cur-

rently operate. Third, because RFID communication is asymmetric, the nodes of an RSN cannot exchange information with each other except through a more powerful reader. However, there are applications for which short-range networks of batteryless sensors would be appropriate; Yeager et al. offer several examples [34].

Computational RFIDs as smartcards. Some passive RFID tags are capable of executing strong cryptographic primitives. For example, various models of the Mifare DESfire can perform triple-DES or AES, while other RFID devices can compute elliptic-curve and RSA signatures, such as the RF360 introduced by Texas Instruments [32]. The RF360 is designed to allow public-key authentication in RFID-enabled identification documents, such as e-passports.

The RF360 incorporates an MSP430, but also includes a cryptographic co-processor, and is designed to operate at relatively short range as a high-frequency, ISO 14443 device. As we show in this paper, CCCP creates the possibility of a more lightweight device. Such a “CCCP smartcard” has two notable benefits: (1) a CCCP smartcard eliminates the cost of cryptography-specific hardware; and (2) a CCCP smartcard can operate in a mode compatible with EPC Gen 2 and achieve read ranges beyond those of a high-frequency device like the RF360.

Some smartcards are capable of performing biometric authentication—generally fingerprint verification. Match-on-card, i.e., verification of the validity of a fingerprint through computation exclusively within the smartcard, has long stood as a technical challenge. The U.S. National Institute of Standards and Technology (NIST) recently conducted an evaluation of a range of such algorithms in contactless cards [11]. CCCP is a promising tool for expanding the class of radio devices for which match-on-card is feasible. While CCCP does

not follow a strict match-in-device paradigm—given that it outsources data to a reader—it nonetheless provides comparable security assurances.

Trusted computing: outsourcing computation via TPMs. CCCP permits a computational RFID to use external memory via an RFID reader. It can support an even broader design space if we use CCCP instead for secure outsourcing not of memory, but of *computational tasks*.

Trusted platform modules (TPMs) [3, 33] offer support for such outsourcing. A TPM is a hardware device, standard in the CPUs of modern PCs and servers, that can provide a secure attestation to the software configuration of the computing platform on which it operates. Briefly stated, an attestation takes the form of a digital signature on a digest of the software components loaded onto the device. (An attestation does not provide assurance against hardware tampering or subversion of running software.)

A computational RFID can in principle make use of a TPM-enabled reader—or platform communicating with the reader—to gain secure access to a more powerful external computer. The process for such use of a TPM is subtle. The operations of verifying a TPM attestation and creating a secure session are both cryptographic operations that require computationally intensive modular exponentiation. Hence the computational outsourcing process requires CCCP as a bootstrapping mechanism.

7 Related Work

CCCP is closely related to Mementos [26] in that both systems provide checkpointing of program execution on CRFIDs. Whereas Mementos relies purely on flash memory and focuses on finding optimal checkpoint frequencies via static and dynamic analysis, CCCP relies primarily on untrusted remote storage via radio and focuses on low-power cryptographic protections to ensure that remotely stored data is as secure as if it were stored locally.

Several systems share CCCP's goal of exploiting properties of RFID systems to enhance security and privacy. For instance, Shamir's SQUASH hash algorithm [31] exploits the underutilized radio link between a tag and a reader to reduce the amount of cryptographic computation necessary on a tag. While number-theoretic hash functions typically require significant computational resources for modular arithmetic, the SQUASH function eliminates costly modular reductions and produces large (unreduced) hash outputs that a tag can send directly to a reader. Tags can thus use the SQUASH function to engage in secure challenge-response protocols with minimal computational resources on the tag. The scheme is provably as one-way as Rabin encryption. Like SQUASH, CCCP exploits the relatively low cost

of radio communication between a tag and a reader to increase security. While SQUASH increases radio communication to reduce computation, CCCP increases radio communication to reduce writes to flash memory.

CCCP uses cryptographic techniques from past work on secure file systems and secure content distribution. CFS [5], the SFS read-only file system [16], and Plutus [22] investigated how to provide secure storage layered on various degrees of untrusted infrastructure. The key generation techniques in secure file systems help CCCP to precompute keystream materials during power seasons. While scalability and throughput are the main challenges in such file systems, CCCP primarily addresses energy and memory constraints. The semantics of CCCP storage are similar to the semantics of secure file systems. None of the systems explicitly and directly prevent denial of service. Storing information on untrusted RFID readers trades off the gain in storage capacity and energy conservation versus the risk of losing data due to compromise or destruction of the external storage. To mitigate the risk against denial of service, CCCP could choose to replicate data as do secure file systems.

CCCP shares some goals with power-aware encryption systems such as that proposed by Chandramouli et al. [10]. Both systems are designed to consume little energy while offering the security of well-known cryptographic primitives and both are motivated by a study of power profiling results, but they have different goals. Chandramouli et al. focus on deriving an energy consumption model and establishing a relationship between energy consumption and security, and they offer an encryption scheme that might allow CCCP to consume less energy during its precomputation of keystream bits. However, CCCP's opportunistic precomputation occurs during periods of abundant energy, when the choice of encryption scheme is not of the utmost importance. CCCP's precomputation allows it to use time- and energy-efficient XOR operations at checkpoint time, when energy is low; an alternative encryption scheme would have to save time or energy over simple XOR operations to be useful when energy consumption matters.

CCCP shares a number of properties with systems built for sensor networks. Storage-centric sensor networks [12, 24] have focused on reducing radio communication and increasing writes to flash memory to conserve energy. One of our motivating observations is that this relationship is inverted in the CRFID model: CCCP reduces writes to flash memory in favor of increasing radio communication. Performing cryptography is hard on both a CRFID and its elder cousin the sensor mote. Previous systems, such as SPINS [25] and TinySec [23] for sensor networks, have faced design choices similar to CCCP's. SPINS and TinySec use RC5 because of its small code size and efficiency, but the battery-powered

platform underlying these systems differs in fundamental ways from batteryless computational RFIDs. For a side-by-side comparison of such embedded systems, see Table 1 of Chae et al. [9].

CCCP provides secure storage for CRFIDs, and CRFIDs are closely related to existing passively powered RFID tags conforming to the EPC Gen 2 standard [13]. At times the RFID and sensor world fuse together. Buetner et al. [7] propose *RFID sensor networks* (RSNs) as a replacement for wireless sensor networks in applications where batteries are inconvenient, and the authors describe RSNs built on WISP CRFIDs. However, the RSN work does not consider remote storage options for CRFIDs.

8 Future Work

Our future work includes enhancements to the CCCP protocol. Most pressing, the protocol currently suffers from a potential atomicity problem. In CHECKPOINT (Algorithm 1), *chkpt_counter* is updated before the checkpointed state is transmitted, so that even if the transmission fails, *chkpt_counter* will point to unused keystream material the next time CHECKPOINT runs. However, if CHECKPOINT updates the offset but terminates before transmission succeeds, then the next RESUME operation will see a value of *chkpt_counter* from which its normal backtracking operation will not find the correct keystream material. CCCP cannot currently recover from such a mismatch.

An unacceptable solution is for CHECKPOINT to update *chkpt_counter* after a successful transmission; such a strategy opens the possibility that, if power loss occurred between the transmission and the counter update, CCCP would reuse keystream material. A more reasonable solution (which we have not implemented) is to use a separate *commit bit* that is set in nonvolatile memory after both the *chkpt_counter* update and the transmission; this solution avoids both problems mentioned above. Minimizing the energy cost of maintaining a commit bit is an opportunity for hardware optimization.

A number of implementation enhancements are also future work. For instance, shortfalls in over-the-air RFID protocols and a lack of drivers on the WISP make the restore procedure unnecessarily complicated and difficult to implement. We also plan to extend the borders of CCCP from checkpointing towards long-term storage as described in Section 3.4.5. Key management makes long-term storage more challenging than checkpointing. Another area for further investigation is modifying the checkpoint function to operate at lower voltages. Writing the counter value to flash memory restricts checkpoints to periods where the available energy can support at least one write to flash memory. Our future work seeks

to circumvent these minimum voltages in order to accomplish secure remote storage for CRFIDs whenever their processors have sufficient energy to compute. Finally, for simplicity, CCCP's communication protocol currently addresses only the scenario in which a single tag communicates with a single reader. We plan to discard that simplifying assumption during further testing in multi-reader infrastructures.

9 Conclusion

CRFIDs enable pervasive computing in places where batteries are difficult to maintain. However, the high energy necessary to erase and write to flash memory makes storage difficult without a constant energy source. CCCP extends Mementos [26] by exploiting the backscatter transmission common on passive RFID systems to remotely store checkpoints on an untrusted RFID reader infrastructure. CCCP protects data with UHF-based MACs, opportunistic precomputation of keystream material for symmetric cryptography, and hole punching to store a counter used to enforce data freshness. Our measurements of a prototype implementation of CCCP on the WISP tag shows that radio-based, remote checkpoints require less energy than local, flash-based checkpoints—despite the overhead of the cryptography to restore the security semantics of local, trusted storage. CCCP gives a CRFID increased storage capacity at low energy cost and enables long-running computations to make progress despite continual power interruptions that destroy the contents of RAM. Moreover, the abstraction provided by CCCP allows application developers to focus on computation rather than space, energy, and security management. Flash memory generally requires a coarse-grained, high-power erase operation before writing a new value. Our hole punching technique allows CCCP to partially reuse unerased flash memory, thus reducing the frequency with which flash memory must be erased.

10 Acknowledgments

We thank Robert Jackson for guidance on RF power consumption; Berk Sunar and Christof Paar for advice on UHF-based MACs; Mike Todd for help with circuit-level aspects of flash memory; Mankin Yuen for assisting with measurements; and John Brattin for identifying algorithmic flaws. We also thank the members of the SPQR group from UMass Amherst CS and ECE for reviewing early drafts of this work. This research was supported by NSF grants CNS-0520729, CNS-0627529, and the NSF REU program. This research is supported in part by UMass through the CVIP Technology Development Fund. This material is based upon work supported by

the U.S. Department of Homeland Security under Grant Award Number 2006-CS-001-000001. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the U.S. Department of Homeland Security.

References

- [1] AHSON, S. A., AND ILYAS, M., Eds. *RFID Handbook: Applications, Technology, Security, and Privacy*. CRC Press, 2008.
- [2] ALIEN TECHNOLOGY. Product Overview: Higgs-3 EPC Class 1 Gen 2 RFID Tag IC, July 2008.
- [3] BERGER, S., CÁCERES, R., GOLDMAN, K. A., PEREZ, R., SAILER, R., AND VAN DOORN, L. vTPM: virtualizing the trusted platform module. In *Proceedings of the 15th USENIX Security Symposium* (2006), USENIX Association.
- [4] BLACK, J., HALEVI, S., KRAWCZYK, H., KROVETZ, T., AND ROGAWAY, P. UMAC: Fast and secure message authentication. In *CRYPTO* (1999), Springer-Verlag, pp. 216–233.
- [5] BLAZE, M. A cryptographic file system for UNIX. In *1st ACM Conference on Communications and Computing Security* (November 1993), pp. 9–16.
- [6] BRASSARD, G. On computationally secure authentication tags requiring short secret shared keys. In *CRYPTO* (1982), pp. 79–86.
- [7] BUETTNER, M., GREENSTEIN, B., SAMPLE, A., SMITH, J. R., AND WETHERALL, D. Revisiting smart dust with RFID sensor networks. In *Proceedings of the 7th ACM Workshop on Hot Topics in Networks (HotNets-VII)* (October 2008).
- [8] CARTER, L., AND WEGMAN, M. Universal hash functions. In *Journal of Computer and System Sciences* (1979), Elsevier, pp. 143–154.
- [9] CHAE, H.-J., YEAGER, D. J., SMITH, J. R., AND FU, K. Maximalist cryptography and computation on the WISP UHF RFID tag. In *Proceedings of the Conference on RFID Security* (July 2007).
- [10] CHANDRAMOULI, R., BAPATLA, S., SUBBALAKSHMI, K. P., AND UMA, R. N. Battery power-aware encryption. *ACM Trans. Inf. Syst. Secur.* 9, 2 (2006), 162–180.
- [11] COOPER, D., DANG, H., LEE, P., MACGREGOR, W., AND MEHTA, K. *Secure Biometric Match-on-Card Feasibility Report*, 2007.
- [12] DIAO, Y., GANESAN, D., MATHUR, G., AND SHENOY, P. Re-thinking data management for storage-centric sensor networks. In *Proceedings of the Third Biennial Conference on Innovative Data Systems Research (CIDR)* (January 2007).
- [13] EPCGLOBAL. EPC Radio-Frequency Identity Protocols, Class-1 Generation-2 UHF RFID. <http://www.epcglobalinc.org/standards/uhf1g2/>, 2008.
- [14] ETZEL, M., PATEL, S., AND RAMZAN, Z. Square hash: Fast message authentication via optimized universal hash functions. In *In Proc. CRYPTO 99, Lecture Notes in Computer Science* (1999), Springer-Verlag, pp. 234–251.
- [15] FONSECA, R., DUTTA, P., LEVIS, P., AND STOICA, I. Quanto: Tracking energy in networked embedded systems. In *8th USENIX Symposium of Operating Systems Design and Implementation (OSDI'08)* (2008), pp. 323–328.
- [16] FU, K., KAASHOEK, M. F., AND MAZIÈRES, D. Fast and secure distributed read-only file system. *ACM Transactions on Computer Systems* 20, 1 (February 2002), 1–24.
- [17] GUTTERMAN, Z., PINKAS, B., AND REINMAN, T. Analysis of the Linux random number generator. In *IEEE Symposium on Security and Privacy* (2006), IEEE Computer Society, pp. 371–385.
- [18] HADDAD, S., CHANG, C., SWAMINATHAN, B., AND LIEN, J. Degradations due to hole trapping in flash memory cells. In *Electron Device Letters* (March 1989), IEEE, pp. 117–119.
- [19] HALPERIN, D., HEYDT-BENJAMIN, T. S., RANSFORD, B., CLARK, S. S., DEFEND, B., MORGAN, W., FU, K., KOHNO, T., AND MAISEL, W. H. Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses. In *IEEE Symposium on Security and Privacy* (May 2008), IEEE Computer Society, pp. 129–142.
- [20] HOMER. *Odyssey*, vol. XI. ca. 750 B.C.
- [21] JUELS, A. RFID security and privacy: A research survey. *IEEE Journal on Selected Areas in Communications* 24, 2 (February 2006), 381–394.
- [22] KALLAHALLA, M., RIEDEL, E., SWAMINATHAN, R., WANG, Q., AND FU, K. Plutus: Scalable secure file sharing on untrusted storage. In *Proc. USENIX Conference on File and Storage Technologies* (San Francisco, CA, December 2003).
- [23] KARLOF, C., SASTRY, N., AND WAGNER, D. TinySec: A link layer security architecture for wireless sensor networks. In *Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems (SenSys)* (November 2004).
- [24] MATHUR, G., DESNOYERS, P., GANESAN, D., AND SHENOY, P. CAPSULE: An energy-optimized object storage system for memory-constrained sensor devices. In *Proceedings of the Fourth ACM Conference on Embedded Networked Sensor Systems (SenSys)* (November 2006).
- [25] PERRIG, A., SZEWczyk, R., WEN, V., CULLER, D., AND TYGAR, J. D. SPINS: Security protocols for sensor networks. *Wireless Networks* 8, 5 (Sept. 2002), 521–534.
- [26] RANSFORD, B., CLARK, S., SALAJEGHEH, M., AND FU, K. Getting things done on computational RFIDs with energy-aware checkpointing and voltage-aware scheduling. In *Proceedings of USENIX Workshop on Power Aware Computing and Systems (HotPower)* (December 2008).
- [27] RANSFORD, B., AND FU, K. Mementos: A secure platform for batteryless pervasive computing, August 2008. USENIX Security Works-in-Progress Presentation.
- [28] RIVEST, R. L. The RC5 encryption algorithm. *Dr Dobbs's Journal—Software Tools for the Professional Programmer* 20, 1 (1995), 146–149.
- [29] SAMPLE, A. P., YEAGER, D. J., POWLEDGE, P. S., MAMISHEV, A. V., AND SMITH, J. R. Design of an RFID-based battery-free programmable sensing platform. In *IEEE Transactions on Instrumentation and Measurement* (2008).
- [30] SCHNEIER, B. *Applied cryptography (2nd ed.): protocols, algorithms, and source code in C*. John Wiley & Sons, Inc., New York, NY, USA, 1995.
- [31] SHAMIR, A. SQUASH—a new MAC with provable security properties for highly constrained devices such as RFID tags. In *Proceedings of the 15th International Workshop on Fast Software Encryption (FSE)* (2008), Springer-Verlag, pp. 144–157.
- [32] TEXAS INSTRUMENTS INCORPORATED. <http://www.ti.com/rfid/shtml/news-releases-11-12-07.shtml>.
- [33] Trusted computing group. <http://www.trustedcomputinggroup.org>.
- [34] YEAGER, D., POWLEDGE, P., PRASAD, R., WETHERALL, D., AND SMITH, J. Wirelessly-Charged UHF Tags for Sensor Data Collection. In *IEEE International Conference on RFID* (2008), pp. 320–327.

Jamming-resistant Broadcast Communication without Shared Keys

Christina Pöpper
System Security Group
ETH Zurich, Switzerland
poepperc@inf.ethz.ch

Mario Strasser
Communication Systems Group
ETH Zurich, Switzerland
strasser@tik.ee.ethz.ch

Srdjan Čapkun
System Security Group
ETH Zurich, Switzerland
capkuns@inf.ethz.ch

Abstract

Jamming-resistant broadcast communication is crucial for safety-critical applications such as emergency alert broadcasts or the dissemination of navigation signals in adversarial settings. These applications share the need for guaranteed authenticity and availability of messages which are broadcasted by base stations to a large and unknown number of (potentially untrusted) receivers. Common techniques to counter jamming attacks such as Direct-Sequence Spread Spectrum (DSSS) and Frequency Hopping are based on secrets that need to be shared between the sender and the receivers before the start of the communication. However, broadcast anti-jamming communication that relies on either secret pairwise or group keys is likely to be subject to scalability and key-setup problems or provides weak jamming-resistance, respectively. In this work, we therefore propose a solution called Uncoordinated DSSS (UDSSS) that enables spread-spectrum anti-jamming broadcast communication without the requirement of shared secrets. It is applicable to broadcast scenarios in which receivers hold an authentic public key of the sender but do not share a secret key with it. UDSSS can handle an unlimited amount of receivers while being secure against malicious receivers. We analyze the security and latency of UDSSS and complete our work with an experimental evaluation on a prototype implementation.

1 Introduction

Due to the shared use of the communication medium, wireless radio communication is not only vulnerable to traditional attacks such as eavesdropping and message synthesis but also to active jamming attacks [2, 20]. In a signal jamming attack, the attacker emits a jamming signal while the legitimate transmission is taking place, thus achieving a denial-of-service (DoS) by blocking, modifying, annihilating, or overwriting the original sig-

nal. Well-known, effective countermeasures against signal jamming attacks are spread-spectrum techniques, in particular Direct-Sequence Spread Spectrum (DSSS) and Frequency Hopping Spread Spectrum (FHSS) [23]. For these techniques to work, the receivers are required to share secret keys with the sender prior to their anti-jamming communication; these keys enable them to derive identical spreading codes or hopping sequences. Shared secrets are also the basis of proposed anti-jamming broadcast schemes [6, 8].

The requirement of pre-shared secret keys, however, imposes limits on the use of common spread-spectrum techniques for anti-jamming communication in scenarios where such secret keys cannot be pre-shared (but which instead rely on, e.g., public-key certificates). This problem (i.e., the lack of techniques for jamming resistance without shared secret keys) was recently observed in [4] and [24] in the context of pairwise communication.

In this work, we focus on a related but different problem for broadcast communication: *How to enable robust anti-jamming broadcast without shared secret keys?* Typical broadcast applications share the need for authenticity and availability of messages that are transmitted by base stations (senders) to a large, unknown number of potentially untrusted (malicious or selfish) receivers. In such settings, a sender communicates to a dynamic set of *trusted* receivers (i.e., the nodes are honest but may be unknown to the sender due to receiver dynamics) or to *untrusted* receivers (which might be interested in obtaining the information themselves but depriving others of it). In both cases, basing the anti-jamming communication on pre-shared keys is not an option because (honest) nodes join the setting *after* the key deployment or because malicious nodes may misuse shared keys for jamming. We can best illustrate this by an example:

A governmental authority needs to inform the public about the threat of an imminent attack. For disseminating information about the risk, a message could contain the level of risk, a timestamp, the physical area of risk,

and the signature of the central authority (CA). Note that if DSSS was used with a (public) spreading code that is known to the attacker or if no spreading was used at all for the transmission, the attacker could easily disrupt the transmission of the message by jamming, thus blocking the propagation of the warning within her transmission radius. The information transferred in this setting is not secret, hence eavesdropping is not considered a risk. What is crucial is the dissemination (broadcast) of *authentic* information to as many receivers as possible within a reasonable timeframe (seconds to few minutes).

As a solution to the described problem, we propose a scheme called *Uncoordinated DSSS* (UDSSS) that enables authentic spread-spectrum anti-jamming broadcast without the requirement of shared secrets. UDSSS follows a similar approach as DSSS, it differs, however, in the following aspect: the spreading code is not pre-defined but chosen by the sender randomly out of a set of publicly available codes. Since no receiver can predict the choice of the sender, UDSSS prevents dishonest receivers from interfering with the communication (to other receivers) while it enables them to obtain the information themselves. After a certain time, every receiver will succeed in identifying the correct spreading code and its synchronization, thus despreading the signal. The required despreading time depends on the coding strategy, the size of the spreading code set, and on the receivers' processing capabilities; we analyze this in detail. Although UDSSS is inherently less efficient than DSSS, it enables broadcast anti-jamming communication in scenarios in which DSSS cannot be used. Besides the example described above, an important application of UDSSS is the jamming-resilient dissemination of navigation signals. As we will show in Section 7, UDSSS enables not only anti-jamming localization for broadcast navigation systems (GPS or similar systems), but it also inherently protects them against a wide range of location-spoofing attacks. We will also show that UDSSS can achieve the same performance as DSSS in the absence of jamming.

In summary, the main contributions of this work are:

- We identify anti-jamming broadcast without shared keys as a relevant problem and we show that it can be addressed using uncoordinated spread-spectrum techniques.
- We propose a scheme called *Uncoordinated DSSS* that supports broadcast anti-jamming communication without shared keys and enables communication in scenarios in which DSSS cannot be used.
- We analyze the performance of UDSSS. We show that a performance comparable to DSSS can be achieved in the absence of jamming and that the expected time for a message transmission to ten receivers takes less than 30 s on state-of-the-art systems under high jamming-probabilities.

- We demonstrate the feasibility of UDSSS by a prototype implementation on a software-defined radio platform [10]; the reception of a typical message takes well below 20 s for 21 dB processing gain on this system. We note that this time can further be significantly reduced on a purpose-built platform (e.g., like the ones used for GPS receivers).

The remainder of the paper is organized as follows: We give background information on DSSS in Section 2 and describe the system and attacker models in Section 3. In Section 4, we present our UDSSS scheme. We analyze its security in Section 5 and its performance in Section 6, including the presentation of our implementation results. In Section 7, we discuss possible applications of UDSSS. Finally, in Section 8, we describe related work and we conclude our paper in Section 9.

2 Background: DSSS

In DSSS, the data signal is modulated with a continuous, pre-defined spreading signal of a higher frequency, also called the *chipping sequence*. During the modulation, the data signal gets spread in the frequency domain and thus becomes resistant against (narrow-band) interference. The resulting signal is modulated (e.g., using phase-shift keying) and – given a sufficiently high frequency of the spreading signal – becomes hidden in the noise of the wireless channel. The processing gain of the communication system (indicating the ratio by which interference can be suppressed relative to the original signal) defines the required length N of the DSSS spreading code, determining the spreading signal. More precisely, given a certain data bit time T_b and a target processing gain defined as $10 \log_{10} \frac{T_b}{T_c}$ in decibel (dB), we get $N = T_b/T_c$, where T_c is the time of a modulated signal chip (a low signal-to-noise ratio requires $T_c \ll T_b$). A typical processing gain of spread-spectrum systems is between 20 dB and 60 dB and results from a chip length $N \in \{100, \dots, 10^6\}$.

In anti-jamming applications, the DSSS spreading signal is secret and shared only by the sender and legitimate receivers. This can be achieved by a shared secret key that is used to seed a pseudo-random generator at the sender and the receivers. The generator outputs a (pseudo-random) chipping sequence which is used to spread the message. In order to despread the signal, the receivers apply a symmetric operation and correlate the received signal with a synchronized replica of the spreading code. Except for the secret code, all other communication parameters (modulation, frequency band, etc.) are public. For the discussion of DSSS we assume that the receivers are synchronized to the sender (later, we will show how we remove this assumption in UDSSS). The synchronization includes both bit and chip time syn-

chronization to the sent signal as well as synchronization with respect to the used spreading code, i.e., the receivers know which code to apply at which point in time in order to despread the received signal. We refer to related literature for a comprehensive discussion of efficient synchronization techniques [2, 20, 23].

In more details, for spreading a message M , the sender uses a *spreading sequence* $c_0 = (c_{0,1}, c_{0,2}, \dots, c_{0,\ell})$ composed of ℓ binary NRZ (non-return to zero) *spreading codes*, $|c_{0,i}| = N$. Typical spreading codes used for DSSS are pseudo-randomly created sequences [23] and codes with well-defined properties such as Walsh-Hadamard [11] or Gold-codes [20]. The sender spreads M by applying code $c_{0,1}$ to the first b bits of M , $c_{0,2}$ to the second b bits and so forth, where b denotes the repetition factor in the use of the spreading codes. By expressing the codes in the time domain, we can define a function $c_0(t) = c_{0,i}[j]$ for $i = \lfloor t/bT_b \rfloor \bmod N$ and $j = \lfloor t/T_c \rfloor \bmod N$, where T_b (T_c) is the data bit (chip) time. The spreading operation can then be written as $d(t) \cdot c_0(t)$, where $d(t)$ is the data signal that carries the message. The sender modulates and transmits the result.

Upon signal reception, each receiver demodulates the signal and samples it (sampling rate $R_s \geq 2/T_c$). It stores the samples in a cyclic buffer which has the capacity to store samples of several message bits, (i.e., for the duration of $T_s = kT_b$, $k > 1 \in \mathbb{N}$). Then, the receiver despreads the data stored in the buffer by computing $\bar{s} := \sum_{i=0}^{T_b R_s} s[i]c_0(t_i)$ for each data bit, where $s[i]$ denotes the i -th value in the buffer and t_i the time when it was sampled. Finally, the result of the bit integration \bar{s} is used to determine the received bit \hat{d}_i . We assume a simple bit decoder that outputs 1 (0) if the integration yields a value greater (lower) than 0 (i.e., $\hat{d}_i = \lceil \bar{s} \rceil$). Finally, after all data bits have been despread, the correctness of the despadding operation is verified by means of the message decoding.

3 System and Attacker Models

3.1 System Model

Our system consists of a sender A and a set of receivers. The goal of the sender is to enable anti-jamming broadcast to the receivers in the presence of communication jamming. We assume that each device is equipped with a radio frontend with transmission and reception capabilities in a corresponding frequency band and that the receivers are computationally capable of efficiently performing (e.g., ECC-based) public-key operations. In addition, each receiver holds an authentic public key of the sender or of the central authority (CA) that can certify the sender's public key. The CA may be off-line at the time of communication.

In our model, P_A denotes the strength of A 's signal arriving at a receiver B ; P_A depends on the strength of the signal sent by A , on the distance between the sender and the receiver as well as on large- and small-scale fading and interference effects [25, 26]. We denote by P_t the minimal required signal strength at the receiver B such that B can successfully decode the signal. In this context, the transmission between A and B in a setting without (active) interference will be successful if *i*) $P_A \geq P_t$, if *ii*) A and B use the same spreading code, and if *iii*) B uses the correct synchronization in its despreading operation (code time and carrier frequency synchronization).

3.2 Attacker Model

We adopt the attacker model from [24] and consider an omnipresent but computationally bounded adversary J with unlimited power supply that is able to eavesdrop and insert messages arbitrarily but can only alter or erase messages by adding her own (energy-limited) signals to the wireless medium; that is, she cannot disable the communication channel by blocking the propagation of signals (e.g., by placing a Faraday cage around a node). The goal of the attacker is to prevent all communication between the sender A and all or some of the receivers. In order to achieve this, the attacker is not restricted to message jamming but can also modify existing or insert new messages. More precisely, the attacker can choose among the following actions:

- She can *jam messages* by transmitting high-power signals that cause the original signal to become unreadable by the receiver. The fraction of the message that the attacker has to interfere with to successfully jam depends on the used coding scheme (e.g., 13% of the message size [16]).
- She can *modify messages* by either flipping single message bits or by entirely overshadowing original messages. In either case, in this attack the messages remain readable by the receiver.
- She can *insert messages* that she generated herself or reuse previously overheard messages. Depending on the signal strength and used spreading codes, the inserted messages might interfere with regular transmissions.

In addition to these types of *attacks*, we follow previous classifications [20] and distinguish different types of *attackers*: static, sweep, random, and reactive jammers. Static, sweep, and random jammers do not sense for ongoing transmissions but jam the channel permanently; they only differ in the regularity of their jamming signals. Reactive jammers initially solely sense for ongoing transmissions and start jamming only after the detection of a message transfer; we express the strength of reactive jammers by their despreading performance $\Lambda_B(N)$,

denoting the number of spreading codes the attacker can apply and check per time unit. Repeater jammers [12] are a subclass of reactive jammers that intercept the signal, low-noise amplify, filter and re-radiate it without requiring or getting knowledge of the used spreading codes. Hybrid jammers are a combination of the above types that jam while searching for message transmissions.

For all attacker types, we assume a finite maximal transmission power and bandwidth. We denote by P_J the maximal signal strength that the attacker is able to achieve at a receiver B ; the attacker can split P_J over an arbitrary number of parallel signal transmissions. Given P_A , the strength of A 's signal at B , we denote by P_j and P_o the minimal required strength of the attacker's signal at B in order to jam or overshadow a message sent from A to B , respectively, provided that the attacker is aware of the used code sequence and its synchronization. We assume $P_t \leq P_A$ and $P_j < P_o$. We further assume that $P_J < \mu P_t$, where μ denotes the number of possible transmissions, i.e., the attacker is not capable of jamming all possible transmissions in parallel; μ depends on the number of available spreading codes and on the attacker's bit and chip synchronization.

4 Jamming-Resistant Broadcast: UDSSS

In this section, we introduce our UDSSS (Uncoordinated DSSS) scheme. UDSSS is an anti-jamming modulation technique based on the concept of DSSS, however, it does *not* rely on pre-shared spreading sequences. In contrast to anti-jamming DSSS communication, where the spreading sequence is secret and shared exclusively by the communication partners, in UDSSS, a public set C of spreading sequences is used by the sender and the receivers. C is not secret and may be known to the attacker. To transmit a message, the sender randomly selects a spreading sequence from the code set and spreads the message with this sequence. The receivers record the signal on the channel and despread the message by applying sequences from C using a trial-and-error method.

The receivers using UDSSS are not time-synchronized to the sender with respect to the spread signal, i.e., they do not know the message bit or chip synchronization. In order to compensate for this (as well as for message losses due to jamming), the sender sends the message repeatedly and the receivers apply a sliding window approach to synchronize to the transmission. The efficiency of UDSSS is therefore determined *i*) by the time that the receivers need to find the right spreading code and its synchronization (we will analyze this in detail in Section 6) and *ii*) by the attacker's jamming success (analyzed in Section 5). Given that, in UDSSS, the receivers need to search through a set of codes and synchronization windows in order to despread the received message,

UDSSS is inherently less efficient than DSSS. However, it provides important advantages over DSSS:

- UDSSS enables anti-jamming communication between nodes that are within each others' transmission ranges but do not share a secret, and
- UDSSS supports broadcast anti-jamming communication for dynamic groups of untrusted receivers.

UDSSS requires the receivers to store all chips received and to analyze them retrospectively to find the used spreading code. The time this takes defines the latency of the communication. The performance and jamming-resistance of UDSSS can be increased by using multiple senders (in contrast to DSSS). More precisely, we consider $m \geq 1$ parallel broadcast transmissions of the same message with *different* spreading codes. This can be achieved by one sender transmitting m signals in parallel—each spread with a different spreading code—or by using m separate sending devices.

In what follows, we describe the details of the UDSSS operations at the sender(s) and the receivers and discuss suitable choices of the UDSSS spreading code set.

4.1 UDSSS Transmission

We envisage *one* sending device, but for generality, our description includes one or multiple senders that transmit the same message in parallel on $m \geq 1$ channels using the code sets C_1, \dots, C_m (not necessarily distinct). For transmitting message M , $|M| \leq \ell$, each sender repeatedly selects a *fresh*, i.e. randomly selected, code sequence $c_s \in C_i$, spreads M using c_s , and transmits the resulting modulated signal. For each transmission a new code sequence is chosen; repeated messages thus get encoded with a different code sequence on each transmission (with high probability). All spreading codes are chosen to be (nearly) orthogonal (strong auto- and low cross-correlations), hence parallel transmissions of multiple senders do not (significantly) interfere with each other; multiple transmissions using the same spreading code and code synchronization can be excluded by agreements between the senders that are, e.g., linked by wires. Each sender repeats the spreading and sending operation either for a well-defined number of iterations (e.g., for emergency alert broadcasting) or continuously (for longer-term applications, e.g. for navigation signals).

Before the UDSSS modulation, each sender applies the following techniques: In order to achieve message authentication, sender A signs the message using its private key SK_A . The sender may also include a timestamp or sequence number in the message in order to achieve replay protection. In order to resist transmission errors, the sender then error-encodes the message before the spreading operation; error-coding makes a message resistant to a certain number of bit errors (e.g., up to 13%

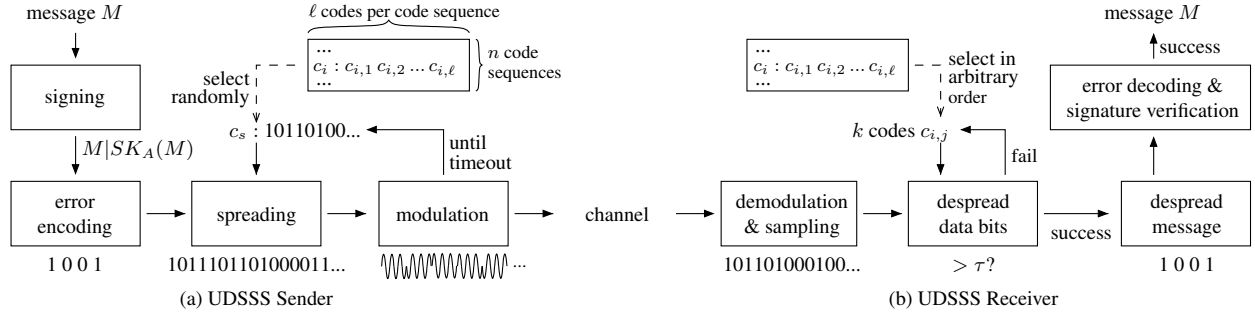


Figure 1: (a) UDSSS transmission. Sender A signs and error-encodes the message M . Then it repeatedly spreads the signed and error-encoded data using a *freshly* selected spreading sequence c_s in each repetition and transmits the modulated signal. (b) UDSSS reception. Receiver B demodulates and samples the radio channel. Then B repeatedly selects a spreading sequence $c_i \in C$, picks k codes $c_{i,j}$ from c_i and tries to despread one data bit (using the integration threshold τ). On success, B despreads the entire message. A failure during the error-encoding check or signature verification restarts the desreading process.

for concatenated Reed-Solomon codes [16]). In combination with bit interleaving, error-encoding increases the resistance of a message to targeted jamming attacks. The entire sending process is displayed in Figure 1a.

There are two reasons why the UDSSS transmission requires message repetitions: *i*) to enable the receivers that are not synchronized to the beginning of the transmission to receive the message and *ii*) due to the risk that the attacker guesses the used code sequence and thus jams the transmission (this risk is also present in DSSS anti-jamming systems). UDSSS receivers will therefore not try to decode all received signals but only those signals that are received in the time intervals when the sender is expected to transmit. For this, the sender either needs to have a (public) transmission schedule (and the receivers need to be precisely time-synchronized to the sender) or the sender has to repeat the transmission of each message such that, when the receivers fill their reception buffers, they will receive the message (Fig. 2).

4.2 UDSSS Reception

In UDSSS, each receiver samples the radio channel (sampling rate $R_s \geq q/T_c$, $q \geq 2$, sample resolution b_s bits) during the sampling period $T_s = sT_M$, and stores the samples in a buffer; T_M denotes the message transmission time and $s \geq 2$ is the number of messages that can be stored in the buffer; given a continuous message transmission, for $s \geq 2$, the signal stored in the buffer will always contain an entire message. After the buffer has been filled, the receiver will reject all signals arriving to its interface (Figure 2) until the message in the buffer is successfully despread and its authenticity is verified.

After the sampling, the receiver tries to decode the data in the buffer by applying a sliding-window pro-

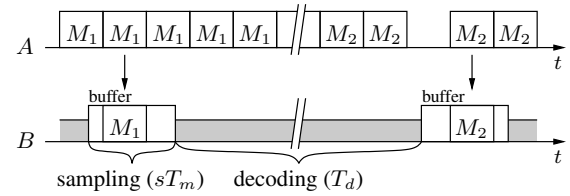


Figure 2: UDSSS message sampling and decoding. A 's repeated transmissions ensure that each receiver B can sample an entire message. After the sampling, B decodes the message M_1 contained in the buffer. During the decoding, B disregards all further samples.

tolocol in which the current window is shifted in intervals of T_c/q ; a complete run of the desreading operation is denoted as one *decoding*. For this purpose, the receiver chooses k spreading codes $c_{i,j}$ ($1 \leq i \leq n$ and $1 \leq j \leq \ell$) from each code sequence $c_i \in C$ (see Figure 3) and uses them to despread k data bits, as sketched in Figure 1b. We check each spreading sequence on multiple ($k > 1$) data bits in order to compensate for transmission or decoding errors. If, during this process and while applying codes from c_r , the absolute value of a bit integration exceeds a threshold τ , i.e. $\bar{s} := \sum_{i=0}^{T_b R_s} s[i]c_{r,j}(t_i) \geq \tau$, the receiver uses the code sequence c_r for desreading the entire message, now benefiting from the identified chip synchronization. τ can be derived from the cross-correlation properties of the used codes and depends on the code length (see Section 4.3). Depending on the available hardware, the desreading operation can partially be performed in parallel or using a multi-stage solution [20].

The bits resulting from this trial-and-error approach are disinterleaved and verified by means of the error-encoding of the message. The receiver accepts those

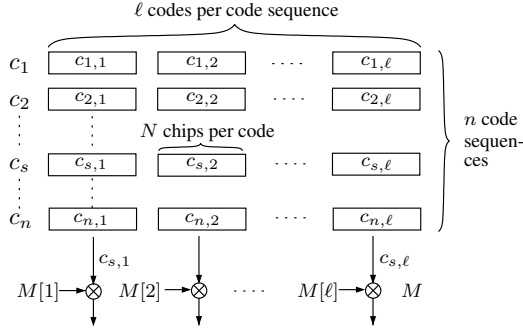


Figure 3: The set C of code sequences. Each sequence $c_i \in C$ is composed of ℓ spreading codes: $c_i = (c_{i,1}, \dots, c_{i,\ell})$, where $|c_{i,j}| = N$. A message M is then spread using a randomly selected code sequence $c_s \in C$; $M[i]$ denotes the i -th bit of M .

messages that pass the error-encoding check and hands them on to the signature verification. Due to possible message insertions by an attacker, the receiver does not stop analyzing the buffer after having successfully despread a message with valid error-encoding, but continues scanning the buffer using the remaining code sequences (until a despread message also passes the signature verification). Thus, the receiver may detect one or more messages per buffer, coming from the original transmissions or from message insertions by the attacker. In any case, the receiver will only pass those messages to the application layer that contain a valid signature.

4.3 UDSSS Spreading Codes

As a crucial component of UDSSS, we now describe how to generate the UDSSS spreading codes that are used by the sender and receivers. In our description, we refer to *one* code set C , however, the same applies for *each* code set in the case of multiple senders. Figure 3 illustrates the code set. Every spreading code $c_{i,j}$ is used to spread *one* bit of the message M (repetition factor $b = 1$), $\ell = |M|$.

UDSSS requires the use of balanced spreading codes that have good auto- and cross-correlation properties; good auto-correlation properties are needed for precise synchronization at the receivers and low cross-correlation properties have the effect that transmissions with different spreading codes do not interfere with each other. We thus exclude the following codes that are typically used in DSSS systems: codes of insufficient length (e.g., Barker codes), codes with large cross-correlation properties (e.g., Walsh-Hadamard), and unbalanced codes resulting in high auto-correlation values (e.g., optical orthogonal codes [7]). Codes for UDSSS that satisfy the above properties are shift-register sequences, in particular *Gold-* and *Kasami-codes*¹ [17],

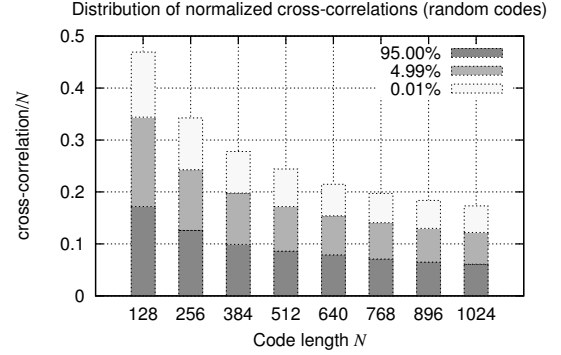


Figure 4: Distribution of cross-correlations for 1000 pseudo-randomly created codes, depending on the code length N . The values are normalized, i.e. divided by N (the peak auto-correlation). The simulations allow to set reasonable limits to the detection threshold τ .

and *pseudo-random sequences* [22].

Due to their straight-forward generation, we focus on pseudo-random codes in the following. A specific code set C is then given by a (public) seed, used as input to a well-defined pseudo-random number generator. Given a sufficiently large code length N , pseudo-random codes have good auto- and cross-correlation properties². Figure 4 displays the cross-correlation values of pseudo-random codes and confirms the desired property; for a more comprehensive analysis of the properties of pseudo-random sequences we refer to [22]. Consequently, the attacker has to use the correct code sequence $c_s \in C$ in order to interfere with a transmission; using a spreading sequence $c' \neq c_s$, $c' \in C$ will not have a relevant impact on the transmission. We can calculate reasonable limits of the parameter ε that specifies the quality of the correlations. Our simulations suggest that, e.g., for random codes of length $N = 512$ (27 dB), $\varepsilon \lesssim 150$ (Figure 4). This enables us to set τ used as integration threshold by the receiver: $\tau = a\varepsilon$, $a \in \mathbb{R} \geq 1$. We refer to Section 6.4 for details on the parameter choices.

Furthermore, the probability that any two random codes of length N from a set of $n\ell$ codes agree is approx. $1 - e^{-(n\ell)^2/2^{(N+1)}}$ (cf. birthday paradox). For typical values of N , n , and ℓ (i.e., $N \geq 64$ and $n\ell \leq 2^{20}$) this probability is negligible. Hence, each code sequence c_i is uniquely identified by any of its codes $c_{i,j}$. While this is beneficial for the legitimate receivers, the attacker will likewise know the entire code sequence if she can successfully identify the code that was used for spreading any particular message bit and might thus be able to jam the remainder of the message. This will be taken into account for the analysis of the attacker's decoding strength (Section 5.2). Section 6.5 will further display the impact of n and N on the system performance.

5 Security Evaluation of UDSSS

In this section, we analyze the points of attack on UDSSS communication and, for various attacker types, derive the probability that a message is jammed during its transmission. As we will show, UDSSS provides resistance even to reactive attackers, a very strong type of attacker.

5.1 Jamming Attacks on UDSSS

An attacker has the following options for performing a code-based jamming attack on UDSSS communication: *i)* she can guess the spreading code and try to jam the signal using this code, *ii)* she can repeat the recorded signal, thus trying to create a collision with the original transmission, and *iii)* she can try to find the code by despreading (part of) the spread signal and then use the identified spreading sequence for jamming the rest of the message *during* its transmission. In the first case, the attacker's jamming signal is independent of the transmission she is trying to jam (representing a static, sweep, or random attacker); in the latter two cases, the attacker is reactive and bases her jamming signal on the detection (and analysis) of the spread signal. In the following we refer to reactive jammers that simply repeat the recorded signal as *repeater jammers* and to reactive jammers that aim at finding the used spreading code as *decoding jammers*. A hybrid jammer can combine non-reactive and reactive actions.

For non-reactive (static, sweep or random) attackers (case *i*), the attacker's success probability depends on the number of codes that she chooses from for composing her jamming signal and on the accuracy of her synchronization to the spread signal. Although (U)DSSS signals are usually hidden in noise, they can be detected by means of energy detectors or by their modulation-specific characteristics [9,20]. Depending on the strength of the attacker and the processing gain achieved by the modulation, the attacker might therefore be able to recover a message transmission and its chip synchronization without having to decode a message; however the attacker still needs to guess the used spreading code in order to jam the signal. In all cases, the jamming success probability of a non-reactive attacker depends on the number of codes in the code set; this probability is further decreased if the attacker cannot detect the chip synchronization (Sec. 5.2).

The purpose of using a different spreading code for each message bit ($b = 1$) is to prevent successful replay attacks from repeater jammers [12] (case *ii*). Due to the low auto-correlation properties of the codes, the attacker's repeated signal would have to arrive at the receiver within one chip length T_c in order to affect the transmission; this requires the attacker to have an (al-

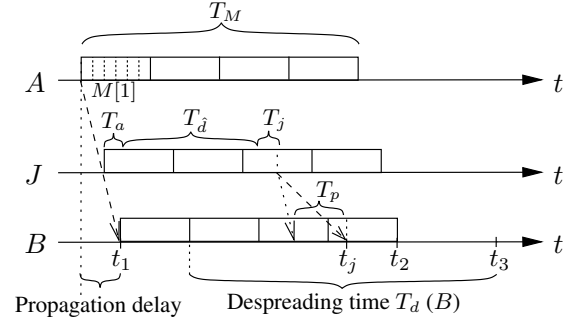


Figure 5: UDSSS attack scenario for reactive (decoding) jammers. Sender *A* sends message *M* with transmission time T_M . Receiver *B* and a reactive jammer *J* start to despread the (same) signal samples after having recorded the first chips. *J*'s jamming attack may only succeed if $t_j < t_2$, i.e., if the attacker succeeds to compose and send its jamming signal such that it reaches *B* before *B* has received the entire message *M*; otherwise the jamming fails and *B* will despread the message at time t_3 . The main advantage for the receiver over the attacker comes from the fact that the attacker only has very short time ($< T_M$) to despread the message, whereas the receiver can despread the message long after having recorded it (within the latency that the application can tolerate).

most) zero processing delay (e.g., for signal inversion) and to be positioned very close to the signal's path of travel (e.g., within a typical $T_c = 10\text{ ns}$, the signal travels less than 3 m). More details are provided in Sec. 5.2.

Although decoding-based attacks (case *iii*) are considered infeasible in DSSS, the probability of such an attack is non-negligible for UDSSS communication due to the restricted number of possible spreading codes. Figure 5 displays the attack scenario for decoding attackers. A decoding attacker needs time to acquire the signal, to detect the spreading code used by the sender, and to exploit this knowledge to compose and transmit the jamming signal. Her reaction time is limited by the message transmission time (T_M) and the fraction that needs to be jammed ($T_{\overline{M}}$). More precisely, the attacker's response time (with effect at the receiver) is composed of:

- T_a time for signal acquisition (min. number of chips)
- T_d expected time for detecting the spreading code
- T_j time for jamming signal generation & transmission
- T_p propagation time difference via the attacker (see Figure 6).

T_a and T_j are mainly determined by the attacker's device, T_d by her computational capabilities (Figure 7), and T_p is given by the attacker's position relative to the sender and receiver. A reactive attacker can be successful with her jamming attack only if $T_a + T_d + T_t + T_p < T_M - T_{\overline{M}}$.

For this reason, the success probability of a decoding

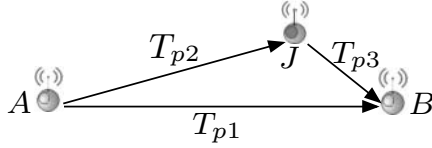


Figure 6: Propagation delay of the jamming signal (for reactive jammers). The displayed times represent propagation delays. $T_p = T_{p2} + T_{p3} - T_{p1}$ is the time that the attacker's signal will be delayed at the receiver B due to propagation delay; $T_p = 0$ if J is positioned on the signal path between A and B .

attacker depends on the time that she needs to identify the used spreading code and its synchronization with respect to the received signal. The code set must limit the search space for the receiver (smaller C is better), while it must still be sufficiently large to prevent the attacker from guessing or systematically finding an effective jamming signal within the message transmission time T_M . Although this might appear as a strong gain in favor of a well-equipped attacker, we stress that the time that the attacker has to find the missing spreading code and its synchronization is small (i.e., limited by T_M , in the order of hundred μs for small messages) while the time for the receiver to despread the recorded message is long (only limited by the application requirements, $\mathcal{O}(s)$). In Section 6, we study how the size of the code set impacts the communication performance of UDSSS.

5.2 Jamming Performance of the Attacker

We now derive the jamming probability for different types of attackers. We use the maximal signal strength P_J that the attacker is able to achieve at the receiver if she transmits with maximal transmission power (Sec. 3.2). Since P_J can be distributed over an arbitrary number of simultaneously transmitted signals, the attacker is allowed to freely choose how much of this power she will use to insert, jam, or overshadow messages as long as the overall signal strength received at B does not exceed P_J . Consequently, given the minimal required signal strength at B such that a message is successfully received (P_t), jammed (P_j), or overshadowed (P_o) (Sec. 3.2), we can derive $n_i := \lfloor \frac{P_J}{P_t} \rfloor$, $n_j := \lfloor \frac{P_J}{P_j} \rfloor$, and $n_o := \lfloor \frac{P_J}{P_o} \rfloor$ as upper bounds for the number of messages that the attacker can insert, jam, and overshadow in parallel.

Static, Sweep, and Random Jamming

We now consider an attacker that tries to guess the used spreading sequence. Let $T_{\overline{M}}$ be the minimum jamming period during which the attacker has to interfere with the transmission of a message M such that it cannot

be decoded by the receiver. The length of this period depends on the used coding scheme: the more bit errors it can tolerate, the longer is $T_{\overline{M}}$. We next compute the probability p_j that a message is jammed for static, sweep, and random jammers (Section 3.2). Sweep and random jammers switch their jamming signal (i.e., the set of code sequences $C_j \subseteq C$ that is jammed) after a duration of $T_{\overline{M}}$ whereas static jammers use the same signal for a time $t \gg T_{\overline{M}}$. Moreover, sweep jammers do not reuse a code sequence until all sequences from C have been used once, whereas random jammers always choose the set C_j at random and might thus select the same code sequences more than once in subsequent jamming attempts. For both the sweep and the random jammer, the number of jamming attempts per message is $T_M/T_{\overline{M}}$. Hence, the probability that a message is successfully jammed by a static jammer is $p_j(n_j) \leq \frac{n_j}{n|M|N}$; for sweep jammers the jamming probability is $p_j(n_j) \leq \min\{\frac{n_j}{n|M|N} \frac{T_M}{T_{\overline{M}}}, 1\}$, and for random jammers it is $p_j(n_j) \leq 1 - (1 - \frac{n_j}{n|M|N})^{T_M/T_{\overline{M}}}$. Note that the attacker has to hit both the right code sequence (out of n sequences) and chip synchronization ($N|M|$).

Reactive and Hybrid Jamming

A reactive decoding jammer tries to find the sender's spreading sequence by performing a search over C . When successful, the attacker knows both the sender's spreading sequence c_s as well as its synchronization and uses this knowledge to jam the remainder of the message M . Throughout this analysis, we make the (worst case) assumption that successfully decoding a single bit of M reveals to the attacker the code sequence that the sender used to spread M (Sec. 4.3). The attacker's ability to jam a message is thus determined by the time that the attacker needs to identify the sender's code sequence and by the time that she then has left to (partially) jam the same message. Let $\Lambda_J(N)$ denote the number of bits that the attacker can despread per second (possibly benefiting from hardware parallelization). The number of code sequences that the attacker is able to verify during the transmission of M such that she detects M 's spreading sequence early enough to be able to successfully jam the message is then $\leq (T_M - T_{\overline{M}})\Lambda_J(N)$. Thus, the probability that a message transmission is detected and jammed is

$$p_j(n_j) \leq \min \left\{ \frac{(T_M - T_{\overline{M}})\Lambda_J(N)}{n|M|N}, 1 \right\}.$$

The despreading performance of a decoding (reactive) attacker is exemplified in Figure 7; in Section 6, we will compare it to the receiver's message decoding performance. Figure 7 shows the expected time that a decoding

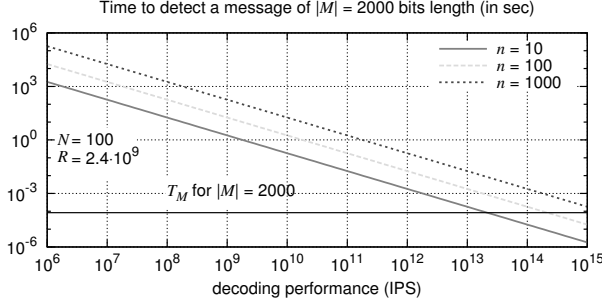


Figure 7: Message detection performance of a decoding attacker as a function of her computing power. This plot depicts the decoding capabilities of a perfect decoding jammer that is able to identify the used spreading code after decoding a single bit (i.e., $k = 1$). The effective impact of the attacker's computing power on the jamming resistance of a message depends on the message transmission time T_M . For a given code set size of $n = 10$ code sequences, in this example, the attacker can block a message of $|M| = 2000$ bits if her computing power exceeds approx. $2 \cdot 10^{13}$ IPS ($2 \cdot 10^{15}$ IPS for $n = 1000$).

attacker (using the receivers' trial-and-error approach) will need to identify the used spreading code sequence; hardware parallelization in the decoding operation can be mapped to a higher decoding performance. The attacker can only be successful if her time to identify the right code sequence is shorter than the message transmission time (intersection with T_M). We point out that even if the attacker uses more elaborate correlation or deconvolution algorithms, her decoding strength can still be expressed by the expected number of bit decodings per second that her algorithm achieves.

Hybrid jammers are a combination of non-reactive and reactive jammers: while searching for the right spreading code, they simultaneously emit a jamming signal. For the most powerful hybrid jammer type, the reactive-sweep jammer [24], the probability that a message is successfully jammed is

$$p_j(n_j) \leq \frac{\eta}{n|M|N} + \left(1 - \frac{\eta}{n|M|N}\right) \min \left\{ \frac{(T_M - T_{\overline{M}})\Lambda_J(N)}{(n - \eta)|M|N}, 1 \right\},$$

where $\eta = \min\{n_j T_M / T_{\overline{M}}, n\}$.

Message Overshadowing: Following the above analysis we can also derive the probability that the transmission of a message is overshadowed by the attacker by substituting n_j with n_o in the above expressions for p_j .

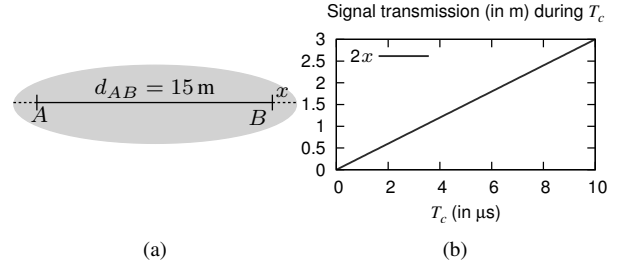


Figure 8: Position of repeater jammers. (a) Area within which a repeater jammer needs to be located in order to successfully jam the communication from A to B (based on signal transmission times). The ellipse is defined by major diameter $d_{AB} + 2x$ and minor diameter $2\sqrt{x^2 + x d_{AB}}$ around A and B; $2x$ is the distance that the signal travels during the chip time T_c ($2x = 3$ m for $T_c = 10^{-8}$ s). (b) Transmission delay ($2x$) of the signal path via the attacker. The smaller T_c , the smaller the area will be in which a repeater jammer needs to be located in order to jam successfully.

Repeater Jammers

As a special case of reactive jammers, *repeater jammers* have $p_j > 0$ only if their signal acquisition, processing, transmission, and propagation delay via the attacker is less than the chip time (i.e., $T_a + T_{pr} + T_j + T_p < T_c$); otherwise the jamming signal will not interfere with the legitimate transmission due to the auto-correlation properties of the spreading codes. For a sample chip rate of 100 Mb/s, the signal travels around 3 m during the transmission of one chip ($T_c = 10^{-8}$ s). This requires the attacker to be positioned within an ellipse with major diameter $d_{AB} + 3$ m around the sender and the receiver in order to jam their communication successfully, see Figure 8. Note that this example considers only the transmission delay of a chip; the attacker's position is even more restricted for $T_a + T_{pr} + T_j > 0$. Hence, repeater jamming implies stringent conditions both on the attacker's position and on her hardware reaction times. Additionally, repeater jamming affects coordinated DSSS and UDSSS equally and we therefore focus on (UDSSS-specific) decoding jammers in the following evaluation.

6 Performance Evaluation of UDSSS

We next evaluate the performance of UDSSS. For simplicity, we first evaluate the scheme for one receiver only and then generalize the results to multiple receivers (Figure 9 displays their decoding performances). We start by analyzing the original UDSSS scheme in the absence of jamming and from that we derive the entire analysis. We will show in Section 6.4 how—in the absence of

jamming—UDSSS can easily be enhanced to yield the same performance as DSSS.

6.1 Communication without an Attacker

In the absence of malicious interference, we can expect that a UDSSS receiver will (on average) successfully decode a message once it has tried a fraction of $\frac{1}{m+1}$ of all codes, where m is the number of parallel transmissions that each use different codes. The expected time for message recovery at the receiver is therefore

$$T_r \approx T_s + T_d = \frac{s|M|N}{R} + \frac{\left(\frac{n}{m+1}Nkq + 1\right)|M|(s-1)}{\Lambda_B(N)}, \quad (1)$$

where $T_s = sT_M$ is the sampling period, $T_M := \frac{|M|N}{R}$ is the time to transmit a message, T_d is the time to decode a message, $R := 1/T_c$ is the chip rate, q is the number of samples per chip, $\Lambda_B(N)$ is the number of bit despreading operations that the receiver B can perform per second (despreading one bit requires Nq additions and multiplications), and k is the number of bits that are despread in order to decide whether the code sequence and synchronization are correct. Thus, the throughput of UDSSS is

$$L = \frac{|M|}{T_r} = \frac{|M|}{\frac{s|M|N}{R} + \frac{\left(\frac{n}{m+1}Nkq + 1\right)|M|(s-1)}{\Lambda_B(N)}} \approx \frac{2\Lambda_B(N)}{nNkq(s-1)}. \quad (2)$$

The approximation holds if $T_s \ll T_d$, that is, if $s|M|N \ll R$ and $1 \ll nN$. For a state-of-the-art system that can execute about 10^{10} IPS, the time T_d to decode a message is in the order of seconds, whereas the time T_M to transmit a message is in the order of hundred μs . In the same setting, DSSS—where the used spreading code and synchronization are known to the receiver—would achieve a throughput of $\frac{|M|}{T_M} = \frac{R}{N}$, which is about one order of magnitude higher than that of UDSSS. However, UDSSS is only used when (coordinated) DSSS cannot be applied for broadcast anti-jamming communication (e.g., due to lack of shared keys). The low throughput of UDSSS should therefore be compared to zero throughput of DSSS. Furthermore, since $\Lambda_B(N) = \mathcal{O}(N^{-1})$ we get $T_r = \mathcal{O}(|M|N^2n)$ and $L = \mathcal{O}(N^{-2}n^{-1})$, showing that increasing the processing gain (i.e., N) is more harmful to the latency/throughput than increasing the code set (i.e., n). Thus, by raising n , an increase of the attacker's processing power can be counteracted with less impact on the message latency than an increase of the attacker's bandwidth and jamming power (which would require raising N).

6.2 Communication in the Presence of an Attacker

We now analyze the impact of message insertion, jamming, and overshadowing on the performance of UDSSS by using the probability p_j (p_o) that a message is jammed (overshadowed), as derived in Section 5.2. Attacker's messages whose signal strengths at the receiver are less than P_j have no impact on regular messages. Consequently, the attacker can insert only up to $n_j := \lfloor \frac{P_j}{P_o} \rfloor$ messages that will interfere with regular message transmissions, provided that they use the same spreading code sequence and synchronization as the sender. The probability that a message inserted by the attacker prevents the successful decoding of a regular message is thus $\leq n_j/(n|M|N)$. Since we assume that all messages are authenticated and integrity-protected with a signature and that the attacker is unable to forge signatures, partially modified messages will be recognized and ignored by the receivers. The only way for the attacker to effectively modify a message is thus to replace it (e.g., by replaying an overheard message).

Let ρ_i , ρ_j , and ρ_o such that $0 \leq \rho_i, \rho_j, \rho_o \leq P_J$ and $\rho_i + \rho_j + \rho_o \leq P_J$ be the power at the receiver that the attacker uses to insert, jam, and overshadow messages, respectively. The expected time to receive a message is then $T_r \leq \mathcal{T}(\lfloor \frac{\rho_i}{P_t} \rfloor, \lfloor \frac{\rho_j}{P_j} \rfloor, \lfloor \frac{\rho_o}{P_o} \rfloor)$, where

$$\begin{aligned} \mathcal{T}(n_i, n_j, n_o) &= \sum_{i=0}^{\infty} p_e^{(s-1)i} (T_s + T_d) = \frac{T_s + T_d}{1 - p_e^{s-1}} \\ &= \left(\frac{sN}{R} + \frac{nNkq(s-1) + sn_i}{\Lambda_B(N)} \right) \frac{|M|}{1 - p_e^{s-1}} \\ &\approx \frac{Nkq|M|(s-1)}{\Lambda_B(N)} \frac{n}{1 - p_e^{s-1}}, \end{aligned} \quad (3)$$

where T_s is the sampling time, T_d the time to decode a message if all codes are tried, and $p_e := (p_j(n_j) + p_o(n_o))^m$; the last approximation holds if $sn_i \leq sn \ll nNkq(s-1)$ and $s|M|N \ll R$.

Theorem 1 (Optimal Choice of the Sampling Buffer Size). *Assuming that the sender is continuously broadcasting the same message, in order to capture the message, the receiver needs to have a buffer capacity of $s = T_s/T_M \geq 2$ messages. In other words, after the sampling, the buffer must contain an entire message for the despreading. Provided that $Nkq \gg 1$, a buffer capacity of $s = 2$ messages is optimal with respect to the expected time to receive a message.*

Proof. Let, by contradiction, $s^* > 2$ be the optimal capacity for the buffer. Hence, from Equation 3, $\frac{(s^*-1)nNkq|M|}{\Lambda_B(N)(1-p_e^{s^*-1})} < \frac{nNkq|M|}{\Lambda_B(N)(1-p_e)}$ must hold, i.e., $\frac{1-p_e^{s^*-1}}{1-p_e} > s^* - 1$. However, for $s^* \geq 2$ we have

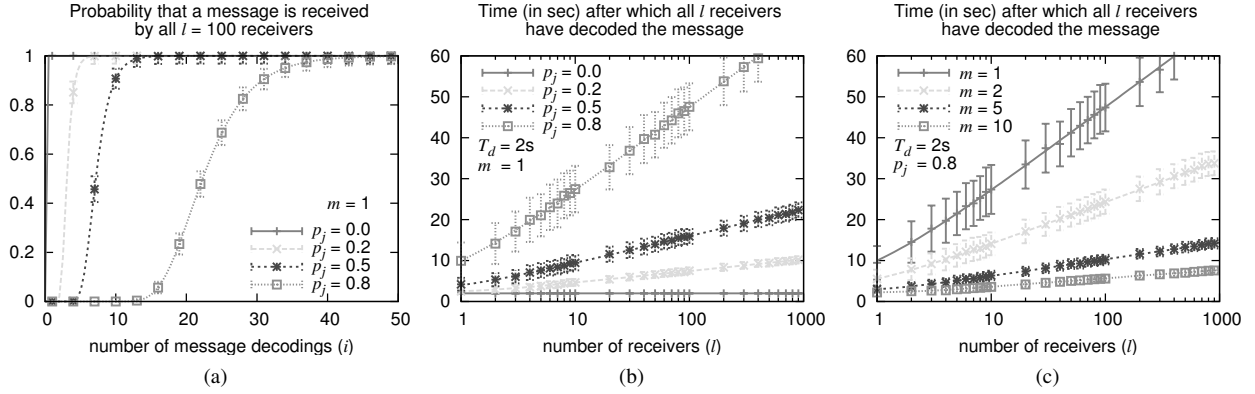


Figure 9: (a) Probability that a UDSSS message has been successfully received by all receivers as a function of the number of message decodings. (b) Expected time to disseminate a message as a function of the number of receivers; the decoding time T_d of the receivers is assumed to be 2 s. (c) Expected time to disseminate a message as a function of the number of parallel message transmissions for a decoding time T_d of 2 s. For (a) – (c), the lines show the expected result according to our analytical analyses, the points and σ -confidence intervals display simulation results.

$\frac{1-p_e^{s^*-1}}{1-p_e} \leq \lim_{p_e \rightarrow 1} \frac{1-p_e^{s^*-1}}{1-p_e} = s^* - 1$, leading to a contradiction. \square

Theorem 2 (Optimal Attacker Strategy). *Given that $Nk \gg 1$, the optimal attacker strategy against UDSSS by which the attacker maximizes the message latency is jamming. That is, for all ρ_i, ρ_j , and ρ_o such that $0 \leq \rho_i, \rho_j, \rho_o \leq P_J$ and $\rho_i + \rho_j + \rho_o \leq P_J$: $\mathcal{T}(\lfloor \frac{\rho_i}{P_J} \rfloor, \lfloor \frac{\rho_j}{P_J} \rfloor, \lfloor \frac{\rho_o}{P_J} \rfloor) \leq \mathcal{T}(0, \lfloor \frac{P_J}{P_J} \rfloor, 0)$.*

Proof. Since $P_j < P_o$ and by definition of p_j and p_o $\forall \alpha_1, \alpha_2 \geq 0 : p_o(\alpha_1) \leq p_j(\alpha_1)$ and $p_j(\alpha_1) + p_j(\alpha_2) \leq p_j(\alpha_1 + \alpha_2)$ it holds that $p_e = p_j(\lfloor \frac{\rho_j}{P_J} \rfloor) + p_o(\lfloor \frac{\rho_o}{P_J} \rfloor) \leq p_j(\lfloor \frac{\rho_j}{P_J} \rfloor) + p_j(\lfloor \frac{\rho_o}{P_J} \rfloor) \leq p_j(\lfloor \frac{\rho_j + \rho_o}{P_J} \rfloor)$. Hence, $\mathcal{T}(\lfloor \frac{\rho_i}{P_J} \rfloor, \lfloor \frac{\rho_j}{P_J} \rfloor, \lfloor \frac{\rho_o}{P_J} \rfloor) \leq \mathcal{T}(0, \lfloor \frac{\rho_i + \rho_j + \rho_o}{P_J} \rfloor, 0) \leq \mathcal{T}(0, \lfloor \frac{P_J}{P_J} \rfloor, 0)$; i.e., spending all power on jamming is optimal for the attacker. \square

6.3 Generalization for Multiple Receivers

If two receivers are synchronized (i.e., sample the same message transmissions) and are positioned appropriately, they will encounter the same attacker-caused errors and require the same amount of time to receive the message (here we assume that the attacker is strong enough to jam all receivers with the same probability, regardless of their relative position to the sender). Moreover, the expected duration $T_r(2)$ until both receivers have successfully received the message equals the single receiver scenario (i.e., $T_r(2) = T_r$). Thus, without loss of generality, any group of receivers that sample the same message transmissions can be regarded as a single receiver.

Now, let l be the number of receivers that sample message transmissions independently (e.g., due to asynchronous sampling schedules, different propagation conditions, or differing distances from the attacker). The probability that at least one of the receivers has not yet successfully received the message once each receiver has sampled i transmissions is $1 - (1 - p_e^i)^l$. Hence, the expected duration $T_r(l)$ until all l receivers have received the message is $T_r(l) \leq \mathcal{T}(n_i, n_j, n_o, l)$, where

$$\begin{aligned} \mathcal{T}(n_i, n_j, n_o, l) &= \sum_{i=0}^{\infty} \left(1 - (1 - p_e^i)^l\right) (T_s + T_d) \\ &\approx \frac{nNkq|M|}{\Lambda_B(N)} \sum_{i=0}^{\infty} \left(1 - (1 - p_e^i)^l\right). \end{aligned} \quad (4)$$

The impact of the number of receivers on the number of required message decodings and on the time to disseminate a message by UDSSS is depicted in Figures 9a and 9b. We observe that even for a high jamming probability of 80%, all receivers have received a message with probability $\geq 90\%$ after about 30 message decodings. Furthermore, the time for all l receivers to receive and decode a message is logarithmic in the number of receivers.

6.4 Optimization and Discussion

One limitation of the UDSSS scheme proposed in Section 4 is its inflexibility to the attacker's strength so that the latency will be high even if no attacker is present. In the following, we analyze techniques to improve the performance of UDSSS. We will show *i*) that selecting a uniform code distribution is optimal and *ii*) that stopping

the decoding process once a valid message was found decreases the message latency. We also show that *iii*) splitting a large code set into smaller, distinct sets for multiple senders does not decrease the message latency in general. For simplicity, we consider one receiver only but the results also hold for multiple receivers.

Theorem 3 (Optimal Code Distribution). *Let $p(c_i)$ denote the probability with which code sequence $c_i \in C$ is selected by the sender. Without loss of generality, let further $1 \geq p(c_1) \geq p(c_2) \geq \dots \geq p(c_n) \geq 0$ and $\sum_s p(c_s) = m$. Selecting c_i under a uniform distribution from a set of n^* codes (i.e., $p(c_i) = m/n^*$ for $1 \leq i \leq n^*$ and $p(c_i) = 0$ for $n^* < i \leq n$) is optimal with respect to the expected time T_r to receive a message.*

Proof. The best strategy for the attacker is to focus her jamming on those codes that are the most likely to be used. Given a code distribution function $p(\cdot)$, $\sum_{i=1}^n p(c_i) = m$, and $\tilde{n}_j = np_j(n_j)$ as the expected number of codes that the attacker can use in parallel to effectively block ongoing transmissions, we get $p_e := \prod_{i=\tilde{n}_j+1}^n (1 - p(c_i))$. It follows from Eq. 3 that T_r is minimized if p_e is minimized, that is, if $p(c_i) = m/n^*$ for $1 \leq i \leq n^*$ and $p(c_i) = 0$ for $n^* < i \leq n$; the optimal number n^* of codes can (numerically) be derived from (3) once $p(c_i)$ and \tilde{n}_j are given. \square

Early Termination at the Receiver. The expected time to receive a message can be reduced if the receiver stops the despreading process once it verified a valid message. Here,

$$T_r \leq \sum_{i=1}^{\infty} p_e^{mi} (T_s + T_d) + \frac{Nkq|M|}{\Lambda_B(N)} \sum_{i=0}^{m-1} p_e^i \frac{n}{m+1} \\ \approx \frac{Nkq|M|}{\Lambda_B(N)} \left(\frac{np_e^m}{1 - p_e^m} + \frac{n}{m+1} \frac{1 - p_e^m}{1 - p_e} \right), \quad (5)$$

where the first term accounts for the number of unsuccessful transmission rounds and the second term is the expected time for the decoding in the last, successful round. Figure 10 compares the expected despreading times of the original UDSSS scheme and the scheme with early termination for multiple senders depending on the jamming probability.

Theorem 4 (Multiple Code Sets). *Consider m sending devices with code sets C_1, \dots, C_m , where $C_i \cap C_j = \emptyset$ for $i \neq j$, $|C_1| \leq |C_2| \leq \dots \leq |C_m|$, and $\sum_i |C_i| = n$, which are broadcasting messages in parallel; the probability for each code sequence $c_j \in C_i$ to be used in the current transmission is $p(c_j) = \frac{1}{|C_i|}$. The expected time T_r to receive a message is equal to the case where the m messages are chosen from one common set C of size n such that $p(c_j) = \frac{m}{n}$.*

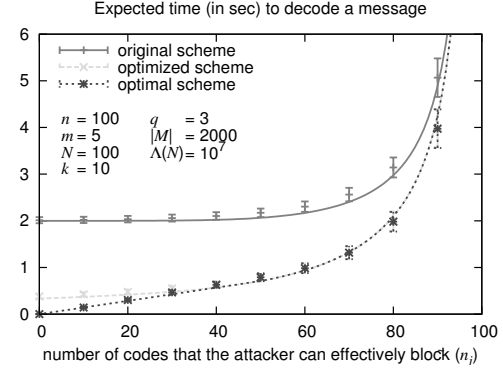


Figure 10: Expected time to disseminate one of the messages from five senders to one receiver. Shown are the original scheme (Equation 3) and the optimization using early termination of the despreading (Equation 5); the optimal scheme uses n^* . We observe that the optimized scheme is close to optimal for $\tilde{n}_j < n/2$ and optimal if $\tilde{n}_j \geq n/2$.

Par.	Definition	Value Range
N	spreading code size	≥ 128 chips (21 dB)
n	#code sequences	up to performance limits
l	#codes per spr. sequence	$ M $
k	#despread data bits per spreading sequence	$2 \leq k \leq \text{\#bits the error-encoding can correct}$
s	buffer size sT_M	2
τ	integration threshold	$N/2$ or 2ϵ

Table 1: UDSSS parameter settings. The larger N and n are, the more jamming-resistant the scheme is. k , s , and τ do not affect the jamming resistance but the decoding performance. A rough, but good estimate for τ is $N/2$; more precise values can be determined by simulations (see Fig. 4), e.g., $\tau = 90$ (200) for $N = 256$ (1024).

Proof. Let a_i denote the number of codes the attacker blocks from the set C_i . The attacker's optimal strategy is to select each a_i such that she maximizes the probability $\tilde{p}_e = \prod_{i=1}^m \frac{a_i}{|C_i|}$ that all m messages are blocked, under the constraints $\sum_{i=1}^m a_i \leq \tilde{n}_j$ and $a_i \leq |C_i| \forall i \in \{1, \dots, m\}$. Hence, \tilde{p}_e is maximized if $\frac{a_i}{|C_i|} = \frac{a_j}{|C_j|}$, i.e., if the attacker jams each code set with the same probability. Then, $|C_i| = \frac{n}{m}$ (Th. 3) and $a_i = \frac{\tilde{n}_j}{m}$, thus $\tilde{p}_e = \prod_{i=1}^m \left(\frac{\tilde{n}_j}{nm} \right) = \left(\frac{\tilde{n}_j}{n} \right)^m$. This probability is equal to the probability $p_e = p_j(n_j)^m = \left(\frac{\tilde{n}_j}{n} \right)^m$ that m messages are blocked if the codes are chosen out of a set of size n where the attacker can block \tilde{n}_j codes. \square

Although splitting a large code set into smaller sets for multiple senders is not beneficial for the latency in general, we can achieve the same message latency as (non-synchronized) DSSS in the absence of jamming by

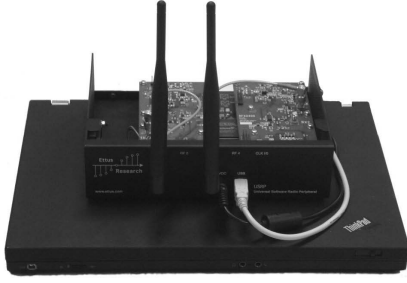


Figure 11: Experimental hardware setup of the UDSSS implementation, consisting of a Universal Software Radio Peripheral (USRP) and a Lenovo T61 ThinkPad.

choosing $m = 2$ with $C_1 = \{c_1\}$, $p(c_1) = 1$, and $p(c_2) = \frac{1}{|C_2|}$. In the absence of jamming, the first code $c_1 \in C_1$ used by the receiver will succeed.

Parameter Selection. The exact UDSSS parameter values depend on the hardware in use and on the assumed attacker strength. The values presented in Table 1 may therefore vary depending on the hardware and application. In general, the product $nN|M|$ represents the security parameter of UDSSS and should at least be in the order of 10^6 ; the smaller $|M|$ is the more jamming-resistant the scheme is.

6.5 Implementation Results

In this section, we demonstrate the feasibility of our UDSSS scheme by means of a prototype implementation based on Universal Software Radio Peripherals (USRPs) [10] and GnuRadio [1] (see Figure 11). The USRPs include a A/D (D/A) converter that provides an input (output) sampling rate of 64 Mb/s (128 Mb/s) and an input (output) sample resolution of 12 bits (14 bits); the employed RFX2400 daughterboards were configured to use a carrier frequency of 2.4 GHz. In our experiments, two USRPs (one being used as a UDSSS sender, the other as a UDSSS receiver) were each connected via a 480 Mbps USB 2.0 link to a Lenovo T61 ThinkPad (Intel Core 2 Duo CPU @ 2.20 GHz) running Linux (kernel 2.6.27) and GnuRadio (version 3.0.3). For performance reasons and for ease of deployment, our UDSSS sender and receiver applications were written entirely in C++, which required porting some GnuRadio libraries from Python to C++. A schematic scheme of our implementation is given in Figure 12.

The sender first encodes the message with a (8,4) Hamming code and scrambles (interleaves) the bits according to a public pseudo-random permutation. Next the sender chooses a spreading code sequence uniform at random, spreads the (encoded and scrambled) message with this code, and sends the resulting chip sequence to the USRP using a differential encoding: the current

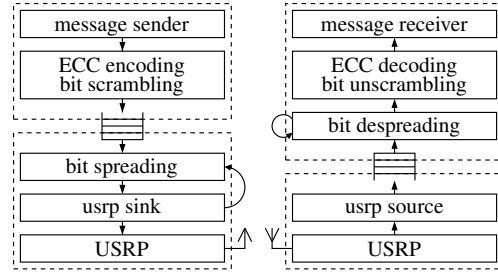


Figure 12: Schematic description of our UDSSS sender and receiver application.

phase of the baseband signal remains unchanged for a $+1$ and its phase is shifted by 180° for a -1 . This step (i.e., choosing a code, spreading, and sending the message) is repeated until the sender stops the message transmission.

The receiver samples the channel for a duration of $2T_M$, where T_M is the transmission time of a message, decodes the samples into a chip sequence, and stores the sequence into a FIFO buffer. A second thread reads the sequences from the FIFO buffer, decodes all possibly included messages by trying all n code sequences on all $N|M|$ positions. To decide whether a code and position pair is valid, a two-level test is used: The sender first despreads two randomly selected bits. If the absolute value of the code-bit correlation for at least one of the bits is $\geq N/2$, it decodes (i.e., despreads, unscrambles, and error-corrects) the first 8 bytes of the message. If these 8 bytes are also valid, the whole message is decoded and the included signature verified.

In our experiments, we positioned the UDSSS sender and receiver indoors at a distance of about 5 m and performed a series of message transfers using UDSSS from the sender to the receiver. The size of the transmitted messages was 256, 512, 1024, 1536, and 2048 bit. The code sets contained up to 500 pseudo-random code sequences and the length of these codes was in the range from 32 to 512 chips. Figures 13a and 13b display the decoding times as a function of the message size $|M|$, code length N , and code set size n . We observe that the decoding time increases linearly with the message size and code set size but quadratic with the code length; this observation is in line with our analytical model. The results further show that, even with this non-optimized (software-based) system, the expected time to receive and decode a typical message ($|M| \leq 2048$ bit) is well below 20 s (for a processing gain of 21 dB and $n = 100$).

We point out that the main purpose of this USRP/CPU-based system is to demonstrate the feasibility of UDSSS. The achieved decoding times should thus not be considered as performance benchmarks. As the operations to decode a bit can easily be executed in parallel, decoding a bit is typically a single-step operation on hardware-based

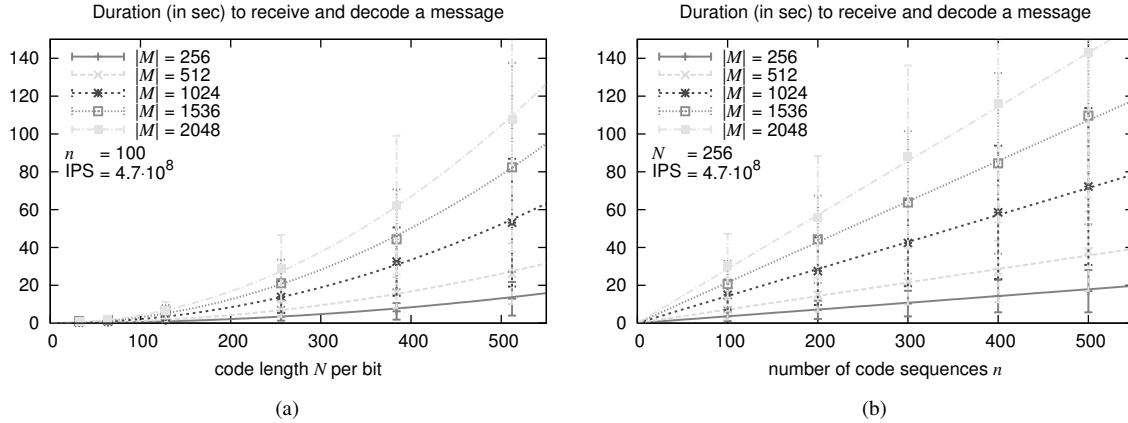


Figure 13: Implementation Results. The plots show the time to receive and decode a message with our UDSSS implementation as a function of the message size $|M|$, code length N , and code set size n . The points and σ -confidence intervals represent the measurements results, the lines the analytical results for a processing speed of about 470 MIPS. We observe that the decoding time increases linear with the message size and code set size but quadratic with the code length. Even with this non-optimized (software-based) system, the expected time to receive and decode a typical message ($|M| \leq 2048$ bit) is well below 20 s (for a processing gain of 21 dB and $n = 100$). On hardware-based DSSS transceivers, bit decoding operations are usually executed in parallel. Purpose-built UDSSS receivers are thus likely to achieve decoding times that are about $O(N)$ times (i.e., 10-1000 times) lower than the times presented in this figure.

DSSS receivers. Realistic decoding times of purpose-built UDSSS transceivers will thus be $O(N)$ times (i.e., 10-1000 times) lower than what we achieve with the presented implementation. As a next step, we intend to optimize our implementation by adding Streaming SIMD Extensions (SSE) support to the core despreading functions and by offloading some of the work to the GPU of the graphic card.

7 Outline of UDSSS Applications

In this section, we present applications for UDSSS broadcasts. The scenarios we will describe share a risk of jamming and of potentially malicious users; in these settings, DSSS communication would either be infeasible or could easily be disrupted by jammers. We demonstrate that the delays which are introduced by the UDSSS trial-and-error reception still enable practical and security-relevant applications.

We consider one or multiple senders that want to disseminate information by broadcasting messages to a set of receivers in a jammed environment. Each receiver holds the authentic public key of the sender but does not share a secret key with it. Such a situation can occur if the sender wants to communicate to a set of *untrusted* receivers that may want to deprive other receivers from obtaining the information broadcasted by the sender, or if a set of *trusted* receivers is dynamic, unknown, or even unpredictable (hence, authentic secret keys between sender and receivers cannot be established beforehand).

Examples for such settings are *i)* *emergency notification (pager) systems* (e.g., Plectron [19]) used to activate emergency response personnel and disaster warning systems or *ii)* *central (governmental) authorities* that need to inform the public about the threat of an imminent or ongoing (terrorist) attack. The danger that attackers jam the alert transmission needs to be minimized. Information dissemination in this setting is clearly time-critical, however, being able to distribute the information within seconds to few minutes is clearly preferred over not being able to disseminate any information at all under jamming. We further argue that, in the absence of jamming, UDSSS permits delays as short as DSSS does (see Section 6.4) and that, once the information has been received by some devices, other communication means (e.g., speech or landline) may additionally support its dissemination to more people concerned.

Another notably well-suited application for UDSSS is the broadcast of *navigation signals* which are foremost used for time synchronization and localization. Examples of navigation systems include satellite navigation (e.g., GPS [27]) and terrestrial systems such as Loran [13] (based on TDoA) and DME-VOR [5] (based on distance/angle measurements). Localization and time-synchronization systems require the reception of navigation signals from multiple base stations; in general, at least three or four different signals are necessary for most localization methods [5]. The broadcast stations are precisely time-synchronized (e.g., via wired links) and located at static or predetermined positions. Each broad-

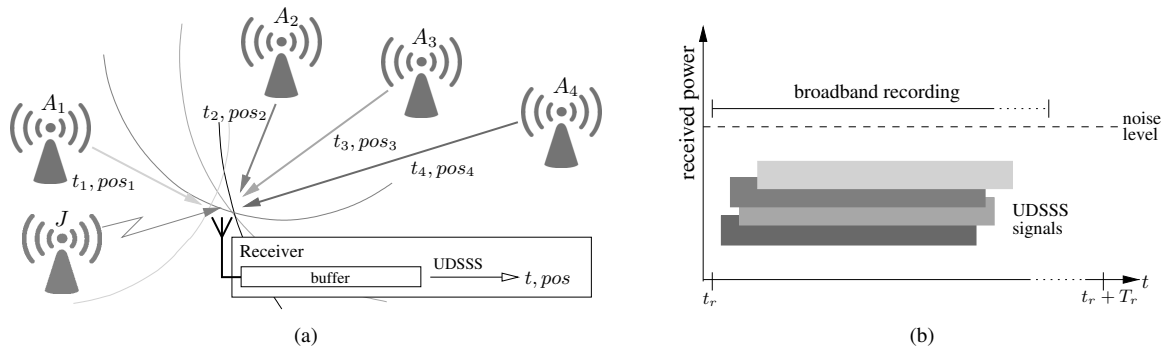


Figure 14: (a) Possible application of UDSSS: jamming- (and spoofing-)resistant reception of navigation signals used for positioning (pos) and/or time-synchronization (t). The receiver records the signals of multiple senders which were spread using randomly selected spreading sequences and uses UDSSS decoding to retrieve the sent messages and compute its position and/or local time. (b) UDSSS signals are highly resistant against narrow-band jamming attacks (by jammer J) because they are sent entirely below noise level. UDSSS likewise prevents signal-delay attacks, because the attacker can only delay individual navigation signals after her decoding, i.e., after having identified the used spreading sequences.

cast station transmits navigation signals either continuously due to a fixed schedule (GPS, Loran-C) or sends replies to individual localization requests (DME-VOR, WLAN-localization), based on which the localized device determines its position.

Without appropriate protection, navigation signals are vulnerable to signal spoofing, synthesis, and jamming attacks [14, 21]. E.g., while current civilian implementations using GPS satellite signals [27] or terrestrial WLAN signals [3] (based on the 802.11b standard) apply spreading to make the transmissions resistant to *unintentional* interference, they do not provide any means to counteract targeted Denial-of-Service (DoS) attacks because their spreading codes are public and can thus be misused for jamming.

UDSSS offers an enhancement to the dissemination of navigation signals that counters targeted jamming. Navigation messages are typically in the order of several hundred bits (e.g., 1.5 kb for GPS messages [18]) and will—even comprising authentication credentials—fit into the considered UDSSS message lengths (in our evaluation in Section 6, the messages were up to 2048 bits long). Each base station uses randomly selected code sequences to spread the messages using UDSSS. The property of the wireless channel enables the receivers to record samples of several navigation signals in parallel in one buffer (same principle as multi-user CDMA). The receivers can then use UDSSS decoding in order to extract three (or more) individual messages (along with their precise arrival times) in one decoding, verify their authenticity, and therefrom derive position and/or time information (see Figure 14a). Unlike DSSS, UDSSS cannot decode navigation signals in real time, but decodes them with a delay T_r , which is largely determined by the process-

ing speed of the receiver. Depending on the implementation and underlying hardware, this delay may vary up to several seconds. However, even if UDSSS causes a delay, the computed position and time are accurate since UDSSS still enables the receiver to record the exact arrival times of the signals it receives.

The delay introduced by the UDSSS decoding is of little importance for pure *time-synchronization* because time represents a rather stable property of a device: Once it is accurately determined, time may slowly degrade by clock drift depending on the clock quality, but it is usually not reset as abruptly as a new position for a mobile device. In this case, after the decoding and processing of the navigation signals, the local time t of the device will be set to $t = t_s + d_p + T_r$, where t_s is the timestamp derived from the base station signals, d_p is the aggregated signal propagation delay (estimated or calculated using the position information, around $30 \mu s$ for 10 km), and T_r is the local time needed for decoding and processing at the receiver (measured time between the first bit filled into the buffer and the moment the time is reset).

So far, we have only discussed the implications of UDSSS on navigation signals in terms of anti-jamming. We now further show that UDSSS equally helps to secure navigation against *spoofing* attacks. In [14], Kuhn showed that time-of-arrival-based navigation systems (like GPS) can be secured against signal-synthesis and selective-replay attacks in which the attacker inserts navigation signals as they would be received at the spoofed location. Without protection, an attacker can manipulate the (nanosecond) relative arrival times by pulse-delaying or replaying of (individual) navigation signals with a delay of Δ , which results in a distance error $c(\delta + \Delta)$ with respect to the true location (where c is the speed of light

and δ accounts for synchronization imprecisions). The asymmetric scheme proposed in [14] is made resistant against these kinds of attacks by decoupling the time-critical signal transmission from a delayed disclosure of the applied spreading code; the first signal is spread and hidden below noise level whereas the second signal (spreading code along with time and position information) is transmitted above the noise level after a delay ρ . A replay attack can now be performed only with a delay $> \rho$. By choosing ρ large enough (e.g., several seconds), even receivers with a low-quality clock can discover the delay in the received timestamps.

UDSSS achieves a similar anti-spoofing protection as the scheme in [14]. Due to the steganographic properties of the UDSSS signal, the attacker can only extract and delay individual navigation signals after having successfully identified the used spreading sequences. Due to a comparison of the received timestamp with the local time, the receiver can identify signal delays that exceed a certain accepted threshold; the threshold basically depends on the accuracy of the receiver's clock. This (probabilistic) approach secures against attacks in which the attacker's decoding takes longer than this threshold.

In contrast to the scheme in [14], which is susceptible to DoS-attacks since data and code are disclosed above noise level, UDSSS provides resistance against jamming because the entire navigation signals are sent with (temporarily) unknown code sequences below noise level (see Figure 14b).

8 Related Work

The impact and detectability of jammers according to their capabilities (e.g., broad- or narrowband) and behavior (e.g., constant, random, reactive) has been widely studied [2, 15, 20, 28]. Spread-spectrum techniques such as DSSS and FHSS are common jamming countermeasures [2, 20]. In [6, 8], the respective authors address broadcast jamming mitigation based on spread-spectrum communication. Additionally, the use of specific coding and interleaving strategies [16] can strengthen the jamming resistance of transmitted messages. Common to these countermeasures is that they all rely on secret keys, shared between the sender and receiver(s) prior to their communication. As argued in prior work [24], pre-loading keys on devices in ad-hoc settings for subsequent jamming-resistant communication suffers from scalability and receiver dynamics problems. Furthermore, if some of the receivers are not trustworthy, relying on pre-shared keys allows malicious receivers to obtain messages themselves while withholding them for others [14].

Recent observations [4, 24] identify the lack of methods for jamming-resistant communication without shared secrets and propose solutions to this problem. The

solution proposed by Baird et al. [4] uses concurrent codes in combination with UWB pulse transmissions. The jamming resistance achieved by their scheme is not one-to-one comparable to common spread-spectrum-based techniques: While the attacker of spread-spectrum techniques must have enough transmission power to overcome the processing gain, in [4] the limiting factor is the number of pulses that the attacker can insert, i.e., her energy. The solution previously proposed based on Uncoordinated Frequency Hopping (UFH) [24] chooses the frequencies of packet transmissions at random from a fixed frequency band. UFH and UDSSS differ significantly in the following aspects: UDSSS is deterministic (apart from the randomness introduced by the attacker) and its performance (the transmission latency) mainly depends on the receiver's processing capabilities. UFH, in contrast, is probabilistic (even in the absence of jamming) and its performance depends on the number of hopping channels (determined by the processing gain). Unlike UFH, UDSSS decouples the processing gain from the spreading uncertainty and allows to fine-tune the scheme (without complex message fragmentations). Finally, due to the unpredictability in the message reception, UFH is unsuitable for applications that require accurate time-stamping of signals, as it is required for many navigation systems.

In [29], an algorithm to estimate the code sequence of a direct spread-spectrum sequence in non-cooperative communication systems is proposed. This algorithm, however, does not leverage the knowledge of the code set used and further assumes that the same code sequence is used repetitively. This approach is therefore not suitable to counter targeted jamming attacks because the communication will no longer be protected once the code sequence has been identified by the attacker.

9 Conclusion

In this paper, we elaborated the problem of broadcast anti-jamming communication without shared secrets, which can, e.g., be used to secure navigation systems. As a solution to this problem we proposed a scheme called Uncoordinated DSSS (UDSSS) that enables DSSS-based broadcast communication *without* pre-shared keys. UDSSS leverages the fact that the sender can transmit a certain amount of spread (hidden) data to the receivers before a (reactive) jammer is able to identify the used code and to jam the transmission. We evaluated the performance and jamming resistance of our DSSS scheme analytically, through a prototype implementation, and by means of simulations for single and multiple receivers. For a state-of-the-art system (about 6000 MIPS), the expected time for a message transfer to a group of 10 receivers takes less than 30 s for a high jam-

ming probability of 80%. We accent that this time is reasonably short, given that with common (key-dependent) anti-jamming techniques the devices would not be able to broadcast jamming-resistant messages at all.

10 Acknowledgments

We are grateful to Fabian Monroe for his valuable input. We also thank the anonymous reviewers for their suggestions. The work presented in this paper was partially supported by the Swiss National Science Foundation under Grant 200021-116444.

References

- [1] GNU Radio Software. <http://gnuradio.org/trac>.
- [2] ADAMY, D. *A first course in electronic warfare*. Artech House, 2001.
- [3] BAHL, P., AND PADMANABHAN, V. N. RADAR: An In-Building RF-Based User Location and Tracking System. In *Proceedings of the IEEE Conference on Computer Communications (InfoCom)* (2000), vol. 2, pp. 775–784.
- [4] BAIRD, L. C., BAHN, W. L., COLLINS, M. D., CARLISLE, M. C., AND BUTLER, S. C. Keyless Jam Resistance. In *Proceedings of the IEEE Information Assurance and Security Workshop* (June 2007), pp. 143–150.
- [5] BENSKY, A. *Wireless Positioning Technologies and Applications*. GNSS Technology and Applications Series. Artech House, 2008.
- [6] CHIANG, J., AND HU, Y.-C. Dynamic jamming mitigation for wireless broadcast networks. In *Proceedings of the IEEE Conference on Computer Communications (InfoCom)* (2008).
- [7] CHUNG, F. R. K., SALEHI, J. A., AND WEI, V. K. Optical orthogonal codes: Design, analysis and applications. *IEEE Transactions on Information Theory* 35, 3 (1989), 595–604.
- [8] DESMEDT, Y., SAFAVI-NAINI, R., WANG, H., CHARNES, C., AND PIEPRZYK, J. Broadcast anti-jamming systems. In *Proceedings of the IEEE International Conference on Networks (ICON)* (1999).
- [9] DILLARD, R. A., AND DILLARD, G. M. *Detectability of Spread-spectrum Signals*. Artech House Publishers, 1989.
- [10] ETTUS. USRP – Universal Software Radio Peripheral. <http://www.ettus.com>.
- [11] GOLDSMITH, A. *Wireless communications*. Cambridge University Press, 2005.
- [12] HANG, W., ZANJI, W., AND JINGBO, G. Performance of DSSS against repeater jamming. In *Proceedings of the IEEE International Conference on Electronics, Circuits, and Systems (ICECS)* (2006), pp. 858–861.
- [13] INTERNATIONAL LORAN ASSOCIATION. LORAN: LOng Range Aid to Navigation. <http://www.loran.org>.
- [14] KUHN, M. G. An asymmetric security mechanism for navigation signals. In *Proceedings of the Information Hiding Workshop* (2004), pp. 239–252.
- [15] LI, M., KOUTSOPOULOS, I., AND POOVENDRAN, R. Optimal jamming attacks and network defense policies in wireless sensor networks. In *Proceedings of the IEEE Conference on Computer Communications (InfoCom)* (2007), pp. 1307–1315.
- [16] LIN, G., AND NOUBIR, G. On link layer denial of service in data wireless LANs: Research articles. *Wireless Communications & Mobile Computing* 5, 3 (2005), 273–284.
- [17] NATARAJAN, B., DAS, S., AND STEVENS, D. An evolutionary approach to designing complex spreading codes for DS-CDMA. *IEEE Transactions on Wireless Communications* 4, 5 (2005), 2051–2056.
- [18] NAVSTAR SPACE AND MISSILE SYSTEMS CENTER. Navstar Global Positioning System: Interface Specification IS-GPS-200. <http://www.losangeles.af.mil>, 2006.
- [19] PARNASS, B. Plectron R-700 Monitor Receivers. *Monitoring Times Magazine* (October 1999).
- [20] POISEL, R. A. *Modern Communications Jamming Principles and Techniques*. Artech House Publishers, 2006.
- [21] RASMUSSEN, K. B., ČAPKUN, S., AND ČAGALI, M. SecNav: secure broadcast localization and time synchronization in wireless networks. In *Proceedings of the ACM International Conference on Mobile Computing and Networking (MobiCom)* (2007), pp. 310–313.
- [22] SARWATE, D. V., AND PURSLEY, M. B. Crosscorrelation properties of pseudo-random and related sequences. In *Proceedings of the IEEE* (May 1980), vol. 68, pp. 593–619.
- [23] SKLAR, B. *Digital communications: fundamentals and applications*. Prentice-Hall, 2001.
- [24] STRASSER, M., PÖPPER, C., ČAPKUN, S., AND ČAGALI, M. Jamming-resistant key establishment using uncoordinated frequency hopping. In *Proceedings of the IEEE Symposium on Security and Privacy* (2008), pp. 64–78.
- [25] TJHUNG, T. T., AND CHAI, C. C. Multitone Jamming of FH/BFSK in Rician Channels. *IEEE Transactions on Communications* 47, 7 (July 1999).
- [26] TSE, D., AND VISWANATH, P. *Fundamentals of Wireless Communication*. Cambridge University Press, 2005.
- [27] U.S. GOVERNMENT. Global positioning system. <http://www.gps.gov>, March 2008.
- [28] XU, W., TRAPPE, W., ZHANG, Y., AND WOOD, T. The feasibility of launching and detecting jamming attacks in wireless networks. In *Proceedings of the ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc)* (2005), pp. 46–57.
- [29] ZHAN, Y., CAO, Z., AND LU, J. Spread-spectrum sequence estimation for DSSS signal in non-cooperative communication systems. In *IEEE Proceedings Communications Magazine, IEEE* (Aug. 2005).

Notes

¹Gold- and Kasami-codes have the same correlation properties and both approach the Welch lower bound in their cross-correlation values. However, Gold- and Kasami-codes differ in the number of codes that can be created. While the number of Gold-codes of length N that can be constructed is $N + 2$ (e.g., 257 for $N = 255$ and 1025 for $N = 1023$), the number of Kasami-codes of length N (in the large set) is considerably higher: $\approx 2^{\frac{3}{2}\log_2(N+1)}$ (e.g., 4112 for $N = 255$ and 32800 for $N = 1023$) [17]. Kasami-codes are therefore more appropriate for UDSSS, although even Kasami codes may have to reoccur in multiple code sequences (if $n\ell > 2^{\frac{3}{2}\log_2(N+1)}$).

² $\forall c_{i,j} \in C, \forall t \in \{0, 1, \dots, N-1\}$, and a small $\varepsilon \ll N$, the auto-correlation of the codes is $\sum_{q=0}^{N-1} c_{i,j}[q]c_{i,j}[q+t \bmod N] \approx N$ if $t = 0$ and $\leq \varepsilon$ else, where $c_{i,j}[q] \in \{-1, +1\}$ denotes the q -th value of the spreading code $c_{i,j}$ and ε indicates the quality of the auto-correlation (the less the better). Similarly, $\forall c_{i,j}, c_{i',j'} \in C, t \in \{0, 1, \dots, N-1\}$ the cross-correlation is $\sum_{q=0}^{N-1} c_{i,j}[q]c_{i',j'}[q+t \bmod N] \leq \varepsilon$.

xBook: Redesigning Privacy Control in Social Networking Platforms

*Kapil Singh**

*School of Computer Science
Georgia Institute of Technology
ksingh@cc.gatech.edu*

*Sumeer Bhola**

*Google
sumeer@acm.org*

Wenke Lee

*School of Computer Science
Georgia Institute of Technology
wenke@cc.gatech.edu*

Abstract

Social networking websites have recently evolved from being service providers to platforms for running third party applications. Users have typically trusted the social networking sites with personal data, and assume that their privacy preferences are correctly enforced. However, they are now being asked to trust each third-party application they use in a similar manner. This has left the users' private information vulnerable to accidental or malicious leaks by these applications.

In this work, we present a novel framework for building privacy-preserving social networking applications that retains the functionality offered by the current social networks. We use information flow models to control what untrusted applications can do with the information they receive. We show the viability of our design by means of a platform prototype. The usability of the platform is further evaluated by developing sample applications using the platform APIs. We also discuss both security and non-security challenges in designing and implementing such a framework.

1 Introduction

Social networking sites have transformed the way people express themselves on the Internet and have become a door to the social life of many individuals. Users are contributing more and more content to these sites in order to express themselves as part of their profiles and to contribute to their social circles online. While this builds up the online identity for the user, it also leaves the data vulnerable to be misused, as an example, for targeted advertising and sale.

More private data online has lead to growing privacy concerns for the users, and some have faced extreme repercussions for sharing their private information on these networking sites. For example, students have been fined for their online social behavior [29]; a mayor was forced to resign because of a controversial Myspace picture [32]. There are numerous such cases, and these incidents clearly underline the importance of privacy control

in social networks.

With the advent of Web 2.0 technologies, web application development has become much more distributed with a growing number of users acting as developers and source of online content. This trend has also influenced social networks that now act as platforms allowing developers to run third-party content on top of their framework. Facebook opened up for third-party application development by releasing its development APIs in May 2007 [22]. Since the release of the Facebook platform, several other sites have joined the trend by supporting Google's OpenSocial [10], a cross-site social network development platform.

These third-party applications further escalate the privacy concerns as user data is shared with these applications. Typically, there is no or minimal control over what user information these applications are allowed to access. In most cases, these applications are hosted on third party servers that are difficult to monitor. As a result, it is not feasible to police the data being leaked from the application *after the data is shared with the application*. There have been several reported cases where users' private information was leaked by the applications, either due to intentional leaks [21] or due to vulnerabilities in the application [26].

Most social networking platforms, such as Facebook, currently provide the applications with full access to user profile information. This permission is granted in Facebook when the user adds the application, which requires the user to make a trust decision. Setting fine-grained access control policies for an application, even if they were supported, would be a complex task. Furthermore, access control policies are not sufficient in enforcing the privacy of an individual: once an application is permitted by a user's access control policy, it has possession of the user's data and can freely leak this information anytime for personal gains. For example, a popular Facebook application, Compare Friends, that promised users' privacy in exchange for opinions on their friends later started selling this information [35].

In this paper, we are concerned with protecting the users' private information from leaks by third-party ap-

*Part of the work was done when the first author was an intern and the second author was an employee at IBM Research T.J. Watson.

plications. We present a mechanism that controls not only what the third-party applications can access, but also what these applications can do with the data that they are allowed to access. We propose and implement a new framework called xBook that provides a hosting service to the applications and enforces information flow control within the framework. xBook provides three types of enforcement that encapsulate the privacy requirements in a typical social network setting: (1) user-user access control (e.g., access to only friends) for data flowing within one application, (2) information sharing outside xBook with external parties; and (3) protection of the application's proprietary data. While (1) and (2) protects the privacy of a user from information leaks, (3) prevents the application's proprietary data or algorithm from being leaked to the application users.

The third-party applications are redesigned in such a way that they have access to all the data they require (allowing them to perform their functionality) and at the same time, not allowing these applications to pass this data to an external entity unless it is approved by the user. *Our framework enforces that the applications make these communications explicit to the user* so that he is more informed before approving an application.

There are several challenges associated with the design of our xBook framework:

Confinement. The execution of application code needs to be confined. This problem needs to be dealt with independently on the client side within the browser and on the server side in the web server. We use “the web server” as a conceptual entity to represent one or more servers.

Mediation. All communication from and within an application needs to be mediated by the xBook platform for permissible information flow. To this end, we developed a labeling model that enforces user-defined security policies. High-level policies specified by the user are converted to low-level labels enforced by xBook.

Programmability. The programming abstraction to the application writers should be practical and easy to use. xBook provides a set of simple APIs in line with the existing social networking platforms.

Portability. The requirements imposed by xBook on the application design should not break the existing applications. In other words, it should be feasible to port most functionality of typical applications to xBook with little effort.

We show the viability of our framework design by implementing a working prototype of our xBook system and porting some of the popular applications from existing social networks, such as Facebook, on top of the framework. We also demonstrate a practical deployment strategy of our system by porting our framework itself as an application on Facebook. Our system is available online [33]. We evaluate the security of our platform by illustrating

some possible application scenarios, and how xBook ensures privacy control in such cases. We also create some synthetic attacks that attempt to exploit the platform to leak information. Our results illustrate that xBook can successfully prevent all such attacks. Our performance results further demonstrate that xBook's privacy control mechanism incurs negligible overhead for typical social networking applications.

The rest of the paper is organized as follows. Section 2 motivates our work by analyzing some privacy issues with the current social networking platforms. We present an overview of our xBook framework in Section 3. Section 4 and 5 discuss the implementation details of xBook's client-side and server-side components, respectively. Our labeling model is described in Section 6. Section 7 presents the evaluation results. We discuss the limitations of our work in Section 8, followed by related work in Section 9. Finally, Section 10 concludes the paper.

2 Background

2.1 Social Networking Platforms

Social networks are the backbone of the online social life of many Internet users. These networks have expanded their development scope by allowing third-party developers to write their own applications, which in turn can be accessed and executed via the social network. An application is an entity that provides some value-added service to the user, and it requires user's profile data to perform its functionality. For example, a simple horoscope application generates daily horoscope based on user's birth information.

Facebook is one popular network that has pioneered the concept of the social network as a platform. The applications bring value both to the platform and its users in providing new features. Applications are deployed on their own servers and Facebook only acts as a proxy for integrating the applications' output to its own pages. The growing popularity of applications on Facebook has enticed other networks, such as Google's Orkut, to start supporting applications. The Orkut platform model is based on the OpenSocial framework [18]. OpenSocial provides a set of APIs for its partner sites (which it refers to as “containers”) to implement. An application that is built for one container should be able to run with few modifications on other partner sites. The APIs allow third parties to have access to the social graph and personal user data.

For the rest of the paper, we use the Facebook case as an example; similar concepts apply to other social networking platforms.

2.2 Privacy Issues with Current Designs

Facebook supports customized policies for user-user access control, but currently provides no control on what

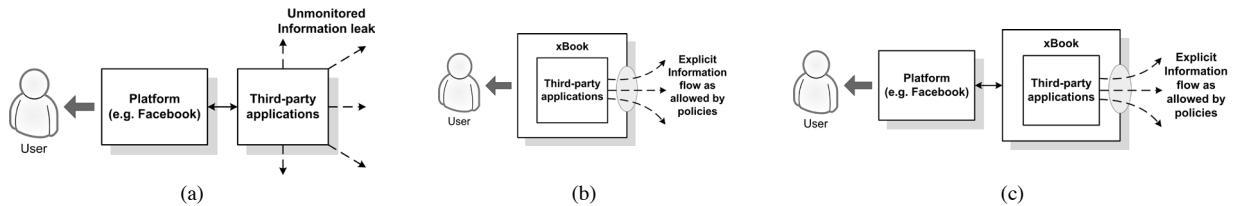


Figure 1: Application architecture for: (a) current platforms. (b) xBook platform. (c) xBook on Facebook.

user profile data can be accessed by third party applications. Applications run on their own servers that have no control administered by Facebook (Figure 1(a)). Applications need data to perform their functionality; they can request user data from the social platform and store it at their own servers. Facebook discourages storing user data on the application’s own servers by barring it in their license agreement [5], but there is no way of enforcing it in Facebook’s current architecture.

Application developers have access to a user’s data even when they are not friends with the user. Unlike a regular friend relationship, this relationship is neither symmetric nor transparent: the application developer has access to the user’s information, but the user does not necessarily know who the application developer is.

Before adding an application, the users are required to agree to a service agreement that allows the application to have access to their profile data. This general agreement is presented for every application, and no other specific information is provided about the application. Since a majority of the applications are known not to exploit users’ personal data, the users tend to add any application, effectively defeating the purpose behind the service agreement. Additionally, second-degree permissions that allow applications to have access to the profiles of the users’ friends add another layer of complexity.

There have been several reported incidents where users’ information was leaked due to a vulnerability in the application [26]. The platform is trusting all third party developers, but the trust is misplaced since there is no restriction on who is allowed to develop an application. One of the most popular Facebook applications, TopFriends, had a vulnerability that allowed any user of TopFriends to see the profile of another user, even if they are not friends with each other [26]. Private information of some high profile users was leaked. Facebook’s response to this controversy was that they *expect* third party applications to follow their policies, which is not acceptable considering that there is no effective way to police the application developers.

User data has a lot of commercial value to marketing companies, competing networking sites, and identity thieves. Therefore, it is not surprising that many applications have been observed to intentionally leak user data to external parties for profit [21]. Other surveys have also discovered similar violations based on an application’s externally-visible behavior [19]. The situation could be

even worse as it is not feasible to determine how many other applications violate the user’s privacy with internal data collection.

Social networking sites have a responsibility to protect user data that has been entrusted to them. The current approach is to legally bind the third parties using a Terms of Service (TOS) agreement [4]. However, it is not possible to monitor the path of information once the information has been released to these parties. Therefore, social networks can not rely on untrusted third parties following their TOS agreements to protect user privacy. Instead, privacy policies should be enforced by the platform and applied to all data that has been entrusted to the social networking site. Our platform design, xBook, is one step forward in this direction.

Felt et al. [19] have proposed a solution to proxy the user information in the form of tags to the third-party applications. These applications do not have access to user data and instead use pre-defined tags to format their output being displayed to the user. Their solution limits the capability of some important and popular applications, such as the horoscope application, that perform processing on user data beyond just displaying it. Our work enforces no such restriction on the application behavior.

3 xBook Overview

xBook is an architectural framework for building social networks that prevents untrusted third-party applications from leaking users’ private information. The applications are hosted on xBook’s trusted platform (Figure 1(b)), and xBook provides complete mediation for all communication to and from these applications.

In a social network setting, an application might communicate with entities outside the xBook system, called *external entities*, to perform specific tasks. For example, the horoscope application may communicate with `www.tarot.com` to receive horoscopes for every sun-sign. The application also encapsulates its own data or algorithm that needs to be protected from untrusted users.

In the xBook framework, applications are designed as a set of *components*; a component being the smallest granularity of application code monitored by xBook. A component is chosen based on what information the component has access to and what external entity it is allowed to communicate with. In the horoscope application, one compo-

nent communicates with `www.tarot.com` and has no access to user data. Another component has access to user's birthday, but does not communicate with any external entity.

From an end user's perspective, the applications are monolithic as the user does not know about the components. At the time of adding a particular application, the user is presented with a manifest that states what user profile data is needed by the application and which external entity will it be sharing this data with. For example, horoscope's manifest would specify that it does not share any information with any external entity. Note that the horoscope application does not need to reveal that it communicates with `www.tarot.com` as no user information is being sent to `www.tarot.com`. The user can now make a more informed decision before adding the application. Admittedly, the user will need to make a trust decision with respect to the parties with which the application shares user data, but these external parties can be expected to be larger and better branded entities providing internet services, such as Google for ads, Yahoo for maps, etc.

Figure 2 shows a typical life cycle of an application. The developer of an application decides on the structure of the components for that application and during the application's deployment on xBook, he specifies the information required by each component and the external entity a particular component needs to communicate with. xBook uses this information to generate the manifest for the application. As shown in the figure, a manifest is basically a set that specifies all of the application's external communications (irrespective of the components) along with the user's profile data that is shared for each communication. Additionally, the xBook platform ensures that all of the application's components comply with the user's privacy policy and the manifest approved by the user. We discuss this further using the case study of an example application in Section 6.3.

The division of an application into multiple components allows the application writer to develop different functionality within an application that rely on different pieces of the user profile. For example, let us consider an application that requires a user's information to generate a customized profile for the user. It also requires his address information to be passed to Google to generate a map showing the address. In the application design of current social networks, the application would be able to pass all information about the user to Google. In the xBook framework, the application would be split into two components: the first component presents the customized profile of the user, has full access to the user's data and is not allowed to communicate with Google; the second component encapsulates the user's address (with no mapping to the user's profile) that is passed to Google to gen-

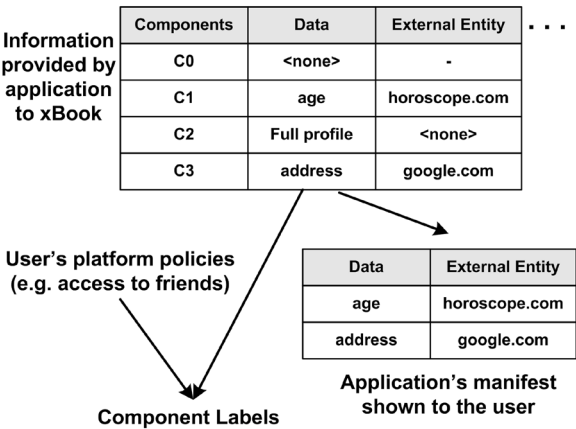


Figure 2: Typical life cycle of an application in xBook.

erate the map. We discuss some example applications in Section 7.1.

Figure 3 shows a high-level design of our xBook framework. There are two parts of the xBook platform, one that runs on the server-side and another that executes on the client-side in the user's browser (Figure 3). The application components, in turn, are also split into client-side and server-side components. The components are written in a safe subset of javascript, called ADsafe [1], which facilitates confinement of these components in our xBook implementation. Any communication to and from the components occurs by using xBook APIs, thereby allowing all such communication to be mediated by xBook. Each component is associated with a privilege level or label that is derived from the application's manifest. The platform mediates the information flow between the components based on these labels (Section 6).

Both client-side and server-side components communicate with server-side storage to retrieve data. There are two types of storage in xBook system: one for storing xBook data that includes user profiles, and second for the data stored by the application. While the structure of xBook data is known, the semantic of the application data is internal to the application and hence unknown to the platform. All data fields are labeled to control access by application components. These labels are assigned based on high-level user-defined policies, such as a policy allowing access to only the user's friends, and the manifest approved by the user (Figure 2).

To store application data with unknown structure and semantics, xBook contains a group of storage pools, where data is stored as a set of name-value pairs. An application can have multiple storage pools, which could be for each user or for user-independent data.

3.1 Leakage Prevention by xBook Design

In the current platform designs, a user's information can be leaked in three major ways: (1) applications can

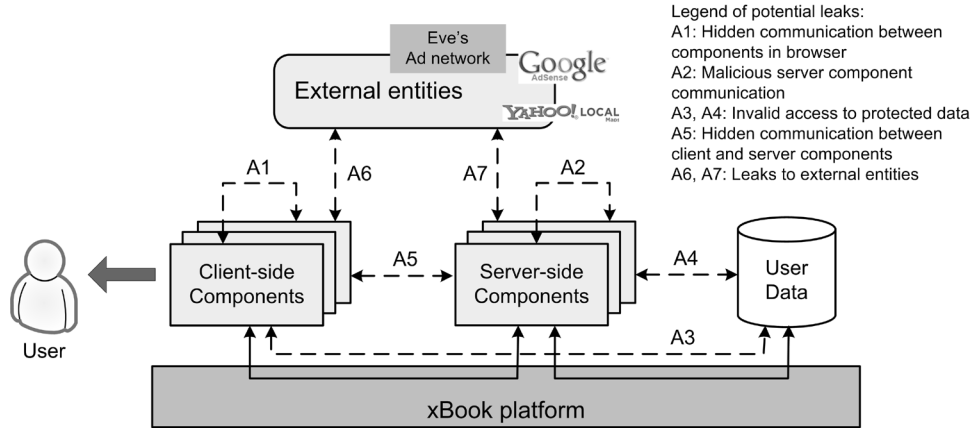


Figure 3: xBook architecture shown along with sources of potential leaks.

share user's information with any third party, including advertisers, or fraudulent parties [21], and as shown in Figure 1(a), there is no way such a leak can be monitored in the current designs; (2) an application can pass information of one application user to another user, breaking free from the platform restriction that only friends can view a user's profile. The reported vulnerability in TopFriends allowed such a leak [26]; (3) the application can recreate the social graph of all its users by connecting common friends as edges in the graph.

xBook's design enforces complete mediation of all communication with the external entities (Figure 1(b)), thus preventing these applications from leaking information, effectively preventing (1) by design. A separate application instance is created for every user, and that instance only has a view of the data accessible to that user. Data access is restricted to allowed user policies, such as access to friends. We mediate any direct or indirect communication between the components of two application instances, thereby deterring (2). (3) is prevented as no single component of an application can have direct access to the data of all its users: a component can only access an anonymized view of this data set (Section 5.2).

xBook, by design, solves most of the leakage problems of the current platforms. However, there are still some potential mechanisms to leak information in our system. We enumerate these possible threats in our formal model and address these threats one by one throughout the paper.

3.2 Formal Requirements

We present a formal model in this section that generalizes xBook's mediation of untrusted third party applications. We use this model to analyze possible attacks, in terms of potential data leaks, under an adversary that deploys an application for collecting users' private data. We also identify a list of requirements that our system should satisfy in order to defeat such attacks. These formal requirements drive the design and architecture of our sys-

tem.

Consider an application A consisting of a set of client-side components and a set of server-side components. Let U be the set of all users of the platform and Y be the set of all external entities. Suppose the application is allowed to communicate to a set of external entities $X \subseteq Y$ and a set of users $F_u \subseteq U$ for a particular user $u \in U$ who is using the system. Now, we divide the set of all data items D into three categories. First, there is a set of proprietary data or code of the application represented as $d_A \subseteq D$. Second, the set of data items $d_{u \rightarrow x}$ belonging to the user $u \in U$ that the application can transfer to the external entity $x \in X$. This set could be in the form of user's age, interests, photos, etc. Third, for an application instance of user $u_i \in U$, the set of data items $d_{u_i \rightarrow u_j}$ is what the application can transfer to a user $u_j \in F_{u_i}$.

The platform wants to monitor the occurrence of a set of events E that can pass information outside an application component. Any event $e \in E$ is actively monitored by intercepting the information flow path between the point of the event occurring and the point where the event is handled. The platform monitors the content information I_e contained in the event. We express the response of the platform when the particular instance of the event has potential leaking information as $R(I_e)$, which may include filtering the content, blocking the communication, etc.

We can identify several sources of potential leaks in the xBook system (Figure 3). The first class of attacks (A1) bypasses the active monitoring by the xBook platform to leak private information from one client-side component to another, by creating a prohibited flow. Such attacks exploit some of the abstract features of the development language and the browser to leak information maliciously. In other words, A1 occurs if response $R(I_e)$ is not triggered even if the I_e contains private information content that is being leaked. Similar leaks (A2) are possible on the server-side where application components can break out

of the sandbox to create a prohibited channel with other components. In addition, some attacks (A3 and A4) can occur during a component's access to data store, where the component gains access to restricted user or application data. Leaks (A5) can also occur in the communication between client-side and server-side components. Other attacks (A6 and A7) leak private information to entities outside the system. The leaks could be to an $x \in Y$ that is prohibited ($x \notin X$), or it could be leaking restricted piece of information $d \in D$ to an entity via communication that is allowed by the system, i.e., for $x \in X, d \notin d_{u \rightarrow x}$ for a user $u \in U$.

We completely forbid cross-application communication, effectively preventing leaks across applications. We also prevent direct communication between server-side components, only allowing them to communicate via storage, thereby preventing attacks of type A2. We mediate other communication paths based on the labels of the communicating parties (Section 6). We address all other identified classes of attacks in Section 7.3. The requirements of an ideal social networking platform that guides the xBook design are as follows:

- Response $R(I_e)$ is invoked if I_e contains prohibited private information. In other words, the platform should be able to monitor any event that might be potentially leaking information, and should take action to prevent such leaks.
- Applications can invoke an event e iff $e \in E$, i.e., applications are restricted to a limited set of events for passing information to external entities.
- Application component having access to user u 's private data d can send information to an external entity $x \in Y$ iff $x \in X$ and $d \in d_{u \rightarrow x}$. In other words, the platform should enforce user policies by limiting the communication to only *allowed* external parties and passing only *allowed* information to these parties.
- Application component having access to user u_i 's private data d can send information to another component acting for user u_j iff $u_j \in F_{u_i}$ and $d \in d_{u_i \rightarrow u_j}$. This means that the applications should inherit the user-user access control policies of the platform.
- Application component x can access d_A only if $x \in S$, i.e., only server-side component of the application should have access to application's proprietary data.

We do not cover attacks against the browser in this work and assume that the browser behaves non-maliciously. Although phishing attacks can entice the user in choosing policies that might leak user information, we do not consider such attacks here. This work enforces the policies specified by the user, and does not consider social engineering attacks against the user.

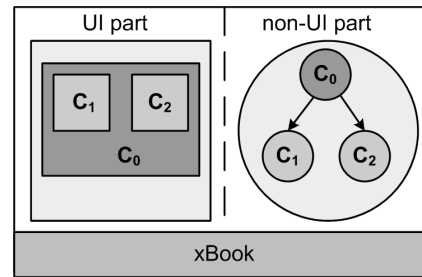


Figure 4: Client-side components in xBook design.

4 Client-side Components

The client-side of the xBook platform and the client components of the applications run within the web browser. The components are further divided into two parts: the user interface (UI) part that is visible as part of the page to the user, and the non-UI part that provides communication interfaces with the external parties and with the server side. There is a one-to-one mapping between the non-UI and the UI parts, i.e., for every non-UI part, there is a corresponding UI part visible to the user (Figure 4).

A component is allowed to create another component. Information can flow during the component creation and this opens up the possibility of an information leak. We prevent such leaks by allowing components to create other components that are at least as restricted as the creating component. This principle prevents the creating component from leaking information out of the system via a less restrictive component.

At the front end, the creating component needs to delegate some screen space to the created component. One challenge is to isolate the third-party application components within the Document Object Model (DOM) of the webpage. A DOM is a platform- and language-independent standard model for representing HTML or XML documents in a browser. We present our confinement approach in the next section.

4.1 Confinement Mechanism

The components of an application encapsulate different levels of private information for the users. Therefore, these components need to be isolated from each other in order to prevent information leaks. On the client side, the components form a part of the DOM of the web page. The web page's DOM may include multiple components from one or multiple applications, apart from the platform's DOM objects.

In the current browser specifications, any script in a page has intimate access to all of the information and relationships of the page. As a result, the components are free to access information about the DOM objects of other components. In order to confine the components within

their own control domain, we limit the application code to be written in an object capability language called ADsafe [1]. In an object capability language, references are represented by capabilities and objects are accessed using these references. Other alternatives to ADsafe, such as Caja [25], are also available; we decided in favor of ADsafe due to its simpler design and easier feature addition and customization to meet our system needs.

ADsafe. ADsafe defines a subset of javascript that makes it safe to put guest code (such as third-party scripted advertising or widgets) on any web page. ADsafe removes features from javascript that are unsafe or grant uncontrolled access to browser elements. Some of the features that are removed from javascript are global variables and functions such as `this`, `eval` and `prototype`. It is powerful enough to allow guest code to perform valuable interactions, while at the same time preventing malicious or accidental damage or intrusion. The ADsafe subset can be verified mechanically by static tools like JS-Lint [8].

ADsafe was initially developed to host untrusted advertising content safely on a webpage. xBook's isolation mechanism is designed with the code base taken from an earlier version of ADsafe. We customized ADsafe by adding code for our component confinement model and mediation based on our labeling model, to prevent information leaks from the "sandboxed" application components. A recent version of ADsafe have since implemented some of our features, but still would require changes to be useful for our system.

One such example is that ADsafe runtime supports only a single level of confinement: all subtrees of the untrusted guest applications exist as children of the trusted web page code. One guest application does not have another guest application as a child to its subtree. In contrast, xBook design requires *nested* DOM subtrees that need to be isolated from each other. Figure 4 shows an example of a nested subtree, where component C_3 is a child of component C_1 , which in turn is a child of C_0 .

Our requirement is to restrict an application component to within a set of connected DOM elements that form the component. In the current DOM specification, any DOM element can parse through the tree of the page via its parent, children or siblings. We enforce confinement by providing the component elements only with a partial view of the page's DOM and only indirect access to the DOM objects.

Confinement Rule 1. One DOM element belonging to an application component should only access another DOM element of the page (that includes accessing its properties, adding a new element to it, etc.) iff they both belong to the same component.

As part of the implementation, xBook associates each component with a unique *DOM wrapper* object at the

time of creation. Figure 5 shows the partial code of our DOM wrapper implementation. Before deploying an application, xBook verifies that each component code is ADsafe compliant. The code must be wrapped in a `<div>` element having an identifier, which forms the root of the component. xBook ensures that this identifier is unique to the application page. The `ADSAFE.go` method gives the component code access to the API object that maps to our DOM wrapper object. The `ADSAFE` code ensures that the second parameter passed to the `createDOMWrapper` function is equal to the identifier of the encapsulating `<div>` element, effectively preventing the developer from faking the identity of the components. It also ensures that the DOM wrapper instance gets the right identity of the component's root node.

The wrapper allows an untrusted component to view DOM nodes simply as integer handles; the component has no direct access to the real DOM. To read or modify the DOM, the component code passes the appropriate handles to the wrapper DOM object using the xBook APIs, which in turn interacts with the real DOM. Additionally, element creation and modification are administered using this component-specific wrapper object. For example, `createTextNode` method in Figure 5 would return an integer handle. Since a wrapper instance is identified by its root element `<div>` that is unique, the DOM wrapper object restricts the untrusted component code to interacting only with the portion of the document tree that belongs to that component. All direct accesses to any real DOM elements are forbidden: the wrapper is the only interface for accessing the elements and it is mediated by the xBook platform.

4.1.1 Event Handling

Another possibility of an application breaking the confinement mechanism originates from the way event handling is designed in the current DOM specification.

Every event has a target, i.e., the XML or HTML element most closely associated with the event. An event handler is a piece of executable code or markup that responds to a particular event. Any element of the DOM can register an event handler to receive a particular event type. Since an event generated from within a component can be received outside the component, the flow of events within a DOM needs to be controlled by the xBook platform for any potential leaks.

In the current DOM implementation, it is possible to assign multiple handlers for a given event. It allows a DOM element to capture events during either of the two phases in the event flow. The event flows down from the root of the document tree to the target element in the first phase called *capture*, then it bubbles back up to the root in the *bubbling* phase. An element can receive the event only if it lies in the path between the document root and

```

function createDOMWrapper(compID, root_node) {
  ...
  /* node2Handle returns the integer mapping of the node */
  /* handle2Node returns the node of the integer handle */
  API.createTextNode = function(str) {
    /* check if str is a string type */
    var node = document.createTextNode(str);
    return node2Handle(node);
  };

  API.appendChild = function(node_handle, child_handle) {
    /* check if node_handle and child_handle are valid */
    var child = handle2node(node_handle);
    handle2node(node_handle).appendChild(child);
  };

  API.addEventListener = function(node_handle,
    eventtype, listenfunction, useCapture) {
    /* check if node_handle is an valid handle */
    handle2Node(node_handle).addEventListener(
      eventtype,
      function(e) {
        /* copy e to new_e while passing only the integer
        handle of the target */
        listenfunction(new_e);
      },
      useCapture);
  };

  API.sendMessage = function(destCompID, message) {
    /* check if destCompID and message are string types */
    /* sendMessage checks validity of information flow
    before passing the message */
    sendMessage(currentUser, compID, destCompID, message);
  };
  return API;
}

ADSAFE = function() {
  /* provides the core ADsafe runtime */
  ...
  return {
    go:function(id, f) {
      /* check if 'id' refers to the <div>
      element (root of the component) */
      var dom = document.getElementById(id);
      if(dom.tagName !== 'DIV')
        error();
      /* create the DOM wrapper and pass its
      reference to the component */
      var API = createDOMWrapper(id, dom);
      f(API);
    }
    ...
  }
}

```

Skeleton ADsafe code added to encapsulate a component code

```

<div id="a0C0">
  <script>
    ADSAFE.go("a0C0", function(API) {
      /* create a button with 'Horoscope' label */
      var elem = API.createElement("button");
      API.appendChild(elem,
        API.createTextNode("Horoscope"));
      ...
      /* send a message to component C1 */
      API.sendMessage("C1", "C0 to C1");
      ...
    });
  </script>
</div>

```

Component Code made ADsafe compliant and verified by JSLint

Figure 5: DOM wrapper implementation with sample functions.

the event target.

One of the goals of our event handling model is to keep the functionality of the current DOM model (including preserving the concept of the two stages). Therefore, we specify our event flow model as follows: for any application component, an element can receive an event iff it lies in the path between the *root of the component* and the target element for the event. We still need to restrict this access to a single component so that no outside component can receive the event; we provide such a restriction by the following confinement rule:

Confinement Rule 2. A DOM element belonging to an application component can receive an event iff the event target belongs to the same component.

We implemented our event handling model using the DOM wrapper object introduced in the previous section. As shown in Figure 5, the object makes a wrapper to the event handling interface available to applications. The wrapper receives the event from the browser's DOM implementation and filters the information presented in the received event object before passing the event to the applications. Any information about the real DOM elements, such as the handler to the target element, is filtered; this prevents application's component code from breaking the confinement. The `addEventListener` method copies the received event `e` into `new_e` while transforming the real DOM element references to wrapped integer values.

The xBook platform mediates the event delivery and as a result, ensures that an event can only be received by elements that belong to the same component that contains the target, thereby enforcing the second confinement rule.

4.2 Communication with External Entities

It is common for the applications to communicate with external parties to perform specific tasks. One typical example is the use of Google map APIs to generate maps of some address known to the application [9]. In other cases, a user's date of birth is used by applications to contact external providers to generate horoscopes [3]. What we achieve in our architecture as compared to the existing social networking platforms is that *we enforce the applications to make these communications explicit* so that more informed decisions can be made. The user or the platform can decide on the policies regarding which external entities are allowed to receive what piece of the user's private information. These policies could be coarse-grained for all applications of a user or fine-grained specific to each application. xBook ensures that the information flows from a specific application component to an external entity according to the defined policies.

There are two kinds of communication flows that can happen in our system:

Symmetric communication in which the response is received by the requesting component. This is a typical case

for most client-server communication in which there is a two-way exchange of information between the two parties.

Asymmetric communication in which the response is not received by the component that made the request, but is handled by another component of the application. Our motivation for supporting this type of communication is to enable some specific application scenarios. One motivating example is the advertising scenario where advertisements are generated by external parties based on the information passed to them: Google generating advertisements based on the address passed to it. These external party advertisements are typically in the form of links that users click to access the related site. If we design this scenario using symmetric communication, these advertising links would not work, since the receiving component has been restricted to communicate only with Google and not any other party. In order to solve this problem, we can create another application component that is considered part of Google's trust domain; since Google servers are unconfined or public from xBook's point of view, the created component is also unconfined. We do not allow any other application component to peek into this new component or disrupt its integrity. Since we are only showing Google's view in this component and the application is not allowed to change this component, this component maintains the trust level of Google. The new component is placed in an `iframe` with its own DOM and hence cannot communicate with any other component. However, since the component is unconstrained, it is allowed to communicate with any external entity and as a result, the advertising links would work.

4.3 Communication between Components: Message Passing Interface

xBook exposes a one-way message passing API that the components use to pass messages to other components. We implement this interface using the DOM wrapper object as shown in Figure 5. The platform mediates this communication and ensures that the information flow model is enforced. Since each component is associated with a unique wrapper object that is used to send the message (Section 4.1), the sending component of the message can not fake its identity to fraudulently pass the information flow checks: as seen in Figure 5, the value of `currentUser` and sender's `compID` are implicitly provided by the wrapper object to xBook's `sendMessage` function. A component can register a message listener with the platform through the xBook API. Any message intended for a particular component is delivered to its message listener. Since the platform knows the identity of each component, it makes sure that the message is delivered to the right component.

The purpose of our message passing interface is to

allow xBook-mediated communication among untrusted components of an application, while still preventing creation of any hidden channels. To this end, we needed to evaluate some of the features of javascript that gives application writers alternatives to pass hidden information in the messages.

Javascript is a weakly typed language and allows any property to be added to any object. For example, an object `message` can take a property `foo` using `message.foo = value`; where `value` could be a number, string or any other object type. Since all application components run in the same scope, a component can pass information to another component if it has access to an object of that component. Let us assume that a component C_1 is allowed to talk to another component C_2 as per the information flow policies, but C_2 can not communicate to C_1 . Effectively, we have a one-way communication channel from C_1 to C_2 . If C_1 passes the object `message` to C_2 , the platform can observe `message`, but cannot identify the object handler `foo` being passed. C_2 can pass information to C_1 by writing to this handler.

We counter such leaks by limiting the message passing to being a JSON container [7], that is pure data. A javascript JSON container is a collection of key/value pairs or an array of values. These key/values are limited to pure data types such as string or numbers. We make a copy of the JSON object and pass the copy to guarantee that there are no additional properties in the passed object. This solution is also effective against attacks by a message sender that use getters and setters.

The simplest way of designing the message passing interface is to pass messages from a source to a destination in a single thread of execution. This option opens up the possibility of a covert communication channel from a more restricted to a less restricted component. For example, let us consider that a less secret component C_0 is passing multiple messages to a more secret component C_1 . Because of the single-threaded non-preemptive nature of javascript, C_1 will complete processing the first message before the control goes back to C_0 . This creates a covert timing channel from C_1 to C_0 . The amount of time taken by C_1 can be observed by C_0 and C_1 can change this time to pass the desired information bits to C_0 .

We reduce the effect of this timing channel by making the message passing interface asynchronous. We achieve asynchronous behavior by implementing a global queue for message passing that is shared among all the components of an application. The receiving components register listeners with the platform in order to receive messages. A timer event dequeues an available message and delivers it to the message listener of the target component of the message. Note that addressing all covert channels in our system is beyond the scope of this paper; we discuss this further in Section 8.

5 Server-side Components

The server-side of the application contains the main functionality for typical applications. It follows a familiar web server model where a server-side component is instantiated for every client request.

Besides the regular user-specific components on the server side, there are certain components that are user independent and works on non-user data or user public data. These components perform two tasks: First, they communicate with external parties to provide functionality independent of the user data. Second, they handle statistical aggregation on user data sets. We discuss declassification based on data anonymization in Section 5.2.

The server components also protect application proprietary data that needs to be declassified before sending it to the client. The threat model is reversed in this case: the applications do not trust the user for their data, so they protect their internal data from being leaked to the users. For example, an application might be giving horoscope predictions to users based on their birth date, but it wants to protect the data or algorithm used for such predictions.

There is no direct communication between the server-side components: all such communication happens via application-specific storage. The platform ensures that the information flow is enforced while accessing the database. The platform also administers the communication with external parties and client-side as allowed by the labeling system.

5.1 Component Confinement

The server-side components need to be isolated from each other. The server-side of xBook mediates all communication flowing in and out from these components. There are several options available for server-side isolation. Operating system isolation mechanisms [12, 30] can be used to sandbox the application components. Another option is a language level confinement similar to the client-side isolation with options like Caja (Javascript) [25], ADsafe (javascript) [1] and JoeE (Java) [20] available. We use ADsafe on the server-side in order to have the same language for developing application components for both client and server.

To the best of our knowledge, we are the first ones to port ADsafe to the server side. We had to make some modification to the ADsafe object to implement our server-side xBook APIs and to perform checking of the information flow labels. Each server-side component holds a unique handle to the modified ADsafe object, and access is restricted to the set of APIs provided by the modified ADsafe object. The modified ADsafe object is conceptually similar to the DOM wrapper object on the client side, but is customized to work in the server-side environment. The platform verifies the validity of the information flow before any access is granted. The javascript ex-

ecution environment is provided by Helma [6], a popular open source web application framework.

5.2 Anonymized Statistics

xBook ensures that no user data is leaked against the user's policies. A particular instance of an application can only have access to profile data that belongs to the user and only his friends. Different instances of the applications cannot share data due to the restrictions posed by xBook's labeling system.

It is desirable for some applications to have a view of all its users so that some statistical results can be published for the whole application. In other words, a component of the application needs to receive data of all the application users and still should be able to share these statistics as output to all users, crossing the boundary of friends.

In order to facilitate this case, we are exploring a three-step anonymization algorithm that provides conservative access to data for the applications. Currently, case 1 and 3 have been implemented, case 2 will be explored as part of our future work.

Case 1. If an application component requests a single field of user information for all application users, it is given access to the requested set in an unmodified form, but in a random order of sequence.

Case 2. If an application component requests multiple fields of user information for all application users, it is given access to the requested set in a form generated by anonymizing the original dataset and then randomizing the resulting tuples' order of sequence. We plan to leverage some of the existing work [15, 24, 31] to generate the anonymized statistics. We acknowledge that providing security in anonymity and statistical queries is a challenging problem and has its own limitations [13, 24]. Addressing these limitations is orthogonal to our work and is not the focus of this paper.

Case 3. Applications can also request the xBook platform for statistics on unanonymized data. This gives the applications more accurate statistics as compared to case 2, where some fields might be filtered or altered to preserve anonymity. xBook provides a limited list of such operations, including aggregation, maximum and minimum value over one or multiple fields.

Discussion. Anonymizing the data might limit some applications that rely on the original data for their functionality. One such example is an application that plots the location of a user's friends on Google maps, and would need to pass names and addresses of the user's friends to Google. The application also makes subsequent queries to Google (for example, to build a Google calendar of friends' birthdays). If the data is anonymized, the application might not produce completely accurate results.

On the other hand, if Google is provided with unanonymized data, it can use the data to cross-reference

and identify the friends. This is a conflict between privacy and functionality. If functionality is preferred and unanonymized information is passed to external entities, user's personal information can be leaked. In such a case, our xBook design, at the minimum, enforces the applications to explicitly declare all external communication (including the data that will be transferred). Based on such information, the user can make a much more informed decision about adding the application.

6 Labeling Model

The xBook platform tracks and enforces information flow using a labeling system defined based on existing models [17,23,27,36]. All system abstractions are layered on top of two types of entities – active and passive. Application components represent active entities that actively participate in label compatibility checks; database entries are passive entities. Every active entity corresponds to a principal and a label; passive entities only have a label.

We do not enforce information flow at the language level [27], but instead at the level of application components and database entries. There are multiple reasons for this choice: (1) it is simpler for the application programmers as they do not need to learn a new language or perform fine-grained code annotations, (2) information flow on a language like javascript with dynamically created source code may not be feasible, and (3) run-time information flow at fine-grained language level would probably be expensive as compared to a much coarser level of components.

The label specifies the secrecy level of an entity. It represents what information is contained in a passive entity and what information the active entity currently has or will read. The entity's principal defines whether the entity has declassification privileges over the label. xBook labels originated along the lines of the language based labels in Jif [27]. Labels represent the confidentiality or secrecy level of an entity in the system. Integrity labeling is not the focus of this work since we are focusing on privacy.

A label L is represented as a set of tags, with each tag having one principal as owner o and another set of principals called readers $R(L, o)$. The owner is the principal whose data was observed in order to construct the data value. The readers represent principals to whom the owner is willing to release the information. An example of a typical label is $L = \{o_1 : r_1, r_2; o_2 : r_2, r_3\}$, where $O(L) = \{o_1, o_2\}$ denote the owner set for the label and readers sets are $R(L, o_1) = \{r_1, r_2\}$ and $R(L, o_2) = \{r_2, r_3\}$.

In the xBook system, principals represent the identities of various entities in the labeling model. There are five types of principals in our system:

- $C(a_i, u_j)$ and $S(a_i, u_j)$ represents the client-side

and server-side components for an application a_i specific to a user u_j .

- $C(a_i)$ and $S(a_i)$ represents user-independent client-side and server-side components for an application a_i .
- u_j represents the entities that the user u_j is in complete control of. Once the user u_j is logged into the xBook system, the user's browser is assigned the principal u_j .
- \top, \perp where \top is highest priority principal in the system and is allotted to the xBook platform. For the sake of completeness, \perp is the least privileged principal.
- External entities also have principal names that contain the hostname and optionally the scheme and port (like in URLs). For example, `https://www.example.com:8888` represents one such principal.

Our model assumes static labels for the entities and information flows from one entity to another if allowed by the label comparison of the end points. Information can flow from one label L_1 to another label L_2 only if L_2 is more *restricted* than L_1 denoted as $L_1 \preceq L_2$.

Restriction. $L_1 \preceq L_2 \iff O(L_1) \subseteq O(L_2)$ and $\forall o \in O(L_1), R(L_1, o) \supseteq R(L_2, o)$

6.1 acts-for Hierarchy

To facilitate easier conversion of user policies to low-level labels, system entities are statically labeled. We decided on immutable labels since it improves usability of the application programming model from the perspective of the application programmer. Unexpected runtime failures can occur when labels of components change at runtime [23]. With immutable labels one can statically verify that all the communication dependencies with respect to other components, external entities, storage will be satisfied.

Some principals have the right to act for other principals and assume their power. The acts-for relation is transitive, defining a hierarchy or partial order of principals [17]. The right of one principal to act for another is predefined by the platform. Figure 6 presents the acts-for relationship within the xBook system. This hierarchy defines the priority of different principles in the system. The reasoning behind the defined hierarchy is as follows:

- \top defines the xbook platform and has the highest security label. As a result, it can declassify any label.
- Any data sink or source that is not explicitly defined by xBook is modeled as an unprivileged entity with label \perp .
- The client-side components are given lower priority than server-side components, because intuitively server-side components residing on xBook servers

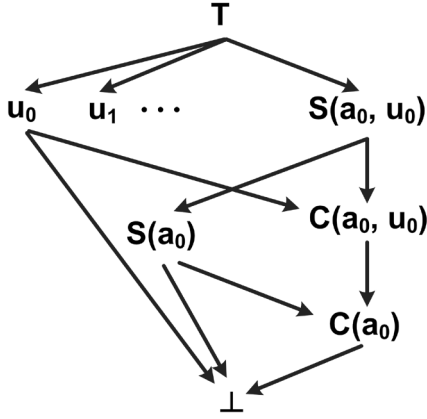


Figure 6: Label hierarchy in xbook.

are more trustworthy than client-side components. For example, $S(a_0, u_0)$ has higher priority over $C(a_0, u_0)$ for application a_0 and user u_0 . The server-side components can declassify an application's proprietary data, which has been labeled in a manner such that it cannot be directly read by client-side components.

- User-independent principals are at a lower priority than any user-specific principal. This allows user-specific components to read user-independent data generated by an application, also effectively allowing users to read statistical data generated for the whole application.
- Principals representing the end user are higher than the corresponding client-side principals since the user controls the client.

6.2 Flow Enforcement

Information flows within the xBook system if the label of source is less restricted than that of destination. Such flow restrictions have been proposed earlier in classical information flow control models [14]. We introduce the concept of endpoints similar to the Flume model [23]. Instead of changing the labels of the entities, for every communication the source and the destination create an endpoint each to facilitate the flow. The entity, based on its principal, can restrict or declassify its label and allocate it to an endpoint for communication. While restricting a label means adding more owners and removing readers, declassification either adds some readers for an owner o or removes the owner o . This relabeling can be done only if the principal of the entity is higher than an owner o in the hierarchy.

Figure 7 shows our flow enforcement algorithm, where maxRestrict and maxDeclassify are defined as:

- **$\text{maxRestrict}(L, P)$.** $O(L) = O(L) \cup \text{descendent}(P)$;
 $\forall o \in \text{descendent}(P): R(L, o) = \{\}$

Algorithm 1 Label Compatibility Check Algorithm.

```

 $eL_1 = (\text{entity}_1 \text{ is a database}) ? L_1 : \text{maxDeclassify}(L_1, P_1)$ 
 $eL_2 = (\text{entity}_2 \text{ is a database}) ? L_2 : \text{maxRestrict}(L_2, P_2)$ 
if  $eL_1 \preceq eL_2$  then
  ALLOW flow from  $\text{entity}_1$  to  $\text{entity}_2$ 
else
  DENY flow
end if

```

Figure 7: Algorithm to check if the information flow from entity_1 to entity_2 is allowed.

- **$\text{maxDeclassify}(L, P)$.** $\forall o \in O(L)$: if $(o \in \text{descendent}(P))$ then $O(L) = O(L) - \{o\}$ where $\text{descendent}(P)$ represents all descendents of a principal P in the acts-for hierarchy, $O(L)$ is the set of owners for label L and $R(L, o)$ represents a set of readers in label L for owner o . Intuitively, the communicating end points support the communication with the sender declassifying its label to the maximum possible using maxDeclassify and the receiver restricting its label using maxRestrict . Since the information can only flow from a less restricted to a more restricted component, these functions facilitate the flow of information.

Some typical flows in the xBook system are depicted in Figure 8. To demonstrate the validity of our algorithm, let us consider the example of the flow between the client-side component C_1 and the server-side component S_1 . For the flow from S_1 to C_1 ,

$$eL_1 = \text{maxDeclassify}(\{S(a_0) ;; \top : C(a_0, u_0)\}, S(a_0, u_0)) = \{\top : C(a_0, u_0)\}$$

$$eL_2 = \text{maxRestrict}(\{\top : C(a_0, u_0)\}, C(a_0, u_0)) = \{C(a_0, u_0) ;; C(a_0) ;; \top : C(a_0, u_0)\}$$

Recollecting the definition of restriction, we can see that $eL_1 \preceq eL_2$, therefore S_1 can send data to C_1 . Considering the reverse flow from C_1 to S_1 ,

$$eL_1 = \text{maxDeclassify}(\{\top : C(a_0, u_0)\}, C(a_0, u_0)) = \{\top : C(a_0, u_0)\}$$

$$eL_2 = \text{maxRestrict}(\{S(a_0) ;; \top : C(a_0, u_0)\}, S(a_0, u_0)) = \{S(a_0, u_0) ;; S(a_0) ;; C(a_0, u_0) ;; (a_0) ;; \top : C(a_0, u_0)\}$$

We can see that $eL_1 \preceq eL_2$, i.e., C_1 can send data to S_1 . Effectively, there is a two-way communication between C_1 and S_1 .

6.3 Case Study: Horoscope Application Lifecycle

An application's lifecycle consists of three steps: the application being hosted by xBook, a user adding the ap-

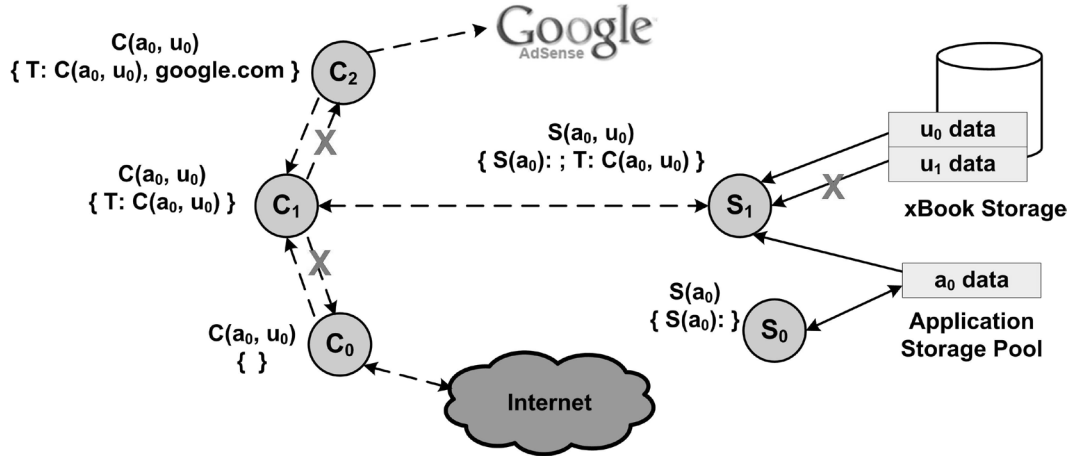


Figure 8: Typical Flows in xBook system with the corresponding labels. For every component, the first parameter is the principal and the second is the label associated with the component.

plication and then the user accessing it.

Hosting. Before xBook accepts a new application, the developer needs to provide the following information:

- The application provides the components to be deployed, in each case specifying if the component is client-side or server-side and if it is user-dependent or not, what user data would the component require and which external entities and other components will it communicate with. In our horoscope example, there are three components: S_0 communicates with `www.tarot.com` and requires no user data; S_1 requires user's birthday; C_1 is on the client-side and also requires user's birthday.
- The application also states that there are user-independent or user-dependent storage pools and each is named declaratively by the application. This ensures that the storage pool names do not leak any user information, as the application has no user information at this time. For example, horoscope application declares a storage pool for storing its application data generated by S_0 .

Based on the label of the user data, xBook derives the labels and the principals of the components. The birthday field has a label $\{T : C(a_i, u_j)\}$, therefore the following labels are allocated to the horoscope components:

- S_0 Principal: $S(a_i)$, Label: $\{S(a_i) : \}$
- S_1 Principal: $S(a_i, u_j)$, Label: $\{S(a_i) : ; T : C(a_i, u_j)\}$
- C_1 Principal: $C(a_i, u_j)$, Label: $\{T : C(a_i, u_j)\}$

The principals define if the component is server-side or client-side, and if it is user-dependent or not. The labels allow S_1 and C_1 to read the birthday field. S_0 's label allows it to declassify itself to be public to communicate with `www.tarot.com`, and write to the storage pool that is given S_0 's label. The storage pool label prevents

any of the client-side components (C_1) from viewing this data, thereby protecting application data from untrusted users. S_1 is allowed to read from the storage pool. The labels of S_1 and C_1 correspond to the labels of S_1 and C_1 respectively in Figure 8, where $i = 0$ and $j = 0$. As we have observed in the last section, the labels of S_1 and C_1 effectively allow a two-way communication channel. Thus, S_1 can pass the results to C_1 that, in turn, can present a formatted form of the horoscope to the user's browser.

Application Addition. When the user is adding the application, he is provided with a manifest that declares what information is passed to which external entity. xBook derives the manifest from the component information provided by the application developer. For example, since none of the components of the horoscope application share any user information with any external entity, horoscope's manifest would specify that it does not pass any information to any external entity. Since the user's birthday is not shared with any external entity, the application does not need to declare its need to access the birthday information.

Application Access. When the user is accessing an application, all user-specific components are instantiated for that user, replacing the user wildcard in the template of labels and principals with the user identifier. This enforces access control across multiple users: access is only granted if it is aligned with the user's privacy policy, for example, access is granted to only user's friends.

7 Evaluation

7.1 Prototype System and Example Applications

We developed a working prototype of the xBook system, which includes platform code and APIs for developing third-party applications. We also implemented the labeling model that enforces information flow control for

Attack Step	Attack Type	Prevented by xBook?
One client component accessing another component's DOM object	A1	✓
Leaks via the message passing interface	A1	✓
A component creates or destroys a less restricted component leaking information	A1	✓
Retrieve information of another user not in the friend list	A3/A4	✓
Client component retrieves more restricted information from the server	A5	✓
Leaks to an unknown external entity	A6/A7	✓
Leaking restricted information to an allowed external entity	A6/A7	✓

Table 1: Prevention of information leaks against various kinds of synthetic attacks.

the data flowing through the system and prevents any information leaks. Our xBook platform consists of about 4300 lines of javascript code.

We developed two sample applications using the xBook APIs to show the ease and viability of application development in xBook. These applications are similar in functionality to two popular Facebook applications: Horoscope [3] and TopFriends [11]. The horoscope application produces a user's daily horoscope based on his birthday information. The utility application based on TopFriends produces a customized profile for the user based on his complete profile information. It also generates a Google map showing the user's home location on the map. The applications are written in javascript using xBook APIs, with the horoscope application having about 180 lines and the application based on TopFriends having around 480 lines of code. We tested these applications against a series of synthetic scenarios, where these applications tried to leak the user's private information. Our tests showed that the xBook system was successful in detecting and preventing all such leaks.

7.2 Porting xBook on Facebook

In order to show the practical viability of the system and to demonstrate that our system can be incrementally deployed, we ported the xBook platform as an application on Facebook. Since Facebook allows any application to have access to user data, including their friends' data, of any user adding the application, xBook as an "application" is able to receive the data of the users agreeing to use the xBook platform. Applications developed using xBook APIs can execute on top of xBook, while still running on xBook servers. Since xBook act as an application for Facebook, xBook's response would be rendered as part of Facebook's web page. Since the third party applications are encapsulated in the page forming xBook's response, the output of these applications would also be effectively rendered on Facebook (Figure 1(c)). Facebook provides the data feed to xBook, which then enables access to this data for xBook applications in a controlled manner through xBook APIs. Facebook's user identity is maintained within xBook. Our running system is available online on Facebook [33].

We envision xBook to be assimilated into the Facebook

platform with Facebook providing two levels of application service. First, the current applications based on current Facebook design would be supported. Second, applications that are developed using xBook APIs are supported, with added privacy protection advantage. Users can be given the discretion to choose between the two options, and the users' choice can drive new application development on xBook.

7.3 Security Analysis

Our analysis shows that xBook prevents the applications from leaking any user information. All of the documented leaks in the current social networks are prevented in the xBook system. For example, the TopFriends leak [26] cannot happen in our system because a separate application instance is created for every user. Each instance only has view of the data accessible to that user and xBook mediates all cross user data accesses.

We evaluated the privacy protection ability of our system in three steps. First, we analyzed the security of the xBook design in view of the potential leaks specified in the formal model (Section 3.2). Second, we developed a set of synthetic attacks targeting the xBook framework and performed experiments to show that our prototype successfully prevents these attacks. Finally, we prove that xBook's information flow model ensures that information leaks cannot happen in the xBook design.

We first analyze the security of our prototype and show that all the attacks discussed in Section 3.2 will not succeed against our design. Attack type A1 is prevented due to the various mechanisms developed in our system for client-side confinement (Section 4.1), such as component isolation, event handling, etc. A2 is prevented by server-side confinement of application components, only allowing them to communicate via storage. Leaks via A3 and A4 are inherently prevented by mediating the information flow from the database to application components with label enforcement based on user-defined policies, and also by anonymizing data for statistical purposes (Section 5.2). A5 is also prevented by label enforcement before the client-side request is passed to the server-side component and before response is returned. Enforcing the confinement model to mediate the external communication, both in synchronous and asynchronous communica-

Application	User latency	Server processing time	Time for label checks (Number of checks)	Overhead
Horoscope	183.1ms	128.8ms	7.7ms (6)	4.2%
Map utility	111.4ms	51.2ms	3.5ms (2)	3.1%

Table 2: Performance results of various operations in typical xBook applications.

tion scenarios, prevents A6 leaks (Section 4.2). Following the same lines, A7 is prevented on the server-side.

Second, we tested the ability of our prototype by creating synthetic exploits that try to break out of xBook’s information flow control model to leak user information. We developed a sample application to launch these attacks against our prototype; if successful, these attacks allow the application to leak information to entities outside the system. Table 1 contains the results of testing our prototype against a wide range of these synthetic attacks. In all our experimental tests, xBook successfully prevented the leaks before the information could be passed outside the system.

We can also prove that if xBook’s confinement mechanism is correctly enforced, the information model ensures that no user information is leaked to external entities (Theorem 1) and to any other user (Theorem 2) outside the user-defined policies.

Theorem 1. *Given a set of policies $P = D \times X$, where the application can pass user’s information field $d \in D$ to external entity $x \in X$, and assuming that the intended confinement is enforced, the information flow model ensures that there is no possible leak outside the xBook system. In other words, if $(d, x) \notin P$ then $\forall C_i : C_i \not\rightarrow^d x$, where C_i are application components and $C_i \rightarrow^d x$ shows that C_i can not pass data item d to x .*

Proof. Let C^0, C^1, \dots, C^k represents the information flow path of a data element d from the xBook database to external entity x .

We present the proof by contradiction. Let us assume that C^i can pass any information (represented by $*$) to x , illustrated as $C^i \rightarrow^* x$. This communication is monitored by our xBook platform, but the platform does not know the semantics of the information being passed.

Also, $\forall i \in [0, k] : C^{i-1} \rightarrow^* C^i \implies L^{i-1} \preceq L^i$ (flow is a restriction)

$C^i \rightarrow^* x \implies L^i \preceq L^x$

Therefore, $L^{i-1} \preceq L^x \implies C^{i-1} \rightarrow^* x$

Continuing this by induction, $C^0 \rightarrow^* x$

In our labeling model, the computational granularity is at the component level. Therefore, we consider that $\forall C_i : \text{Output}(C_i) = F(\text{Input}(C_i))$ for any computation F .

For component C^0 , $\text{Input}(C^0) = d$, $\text{Output}(C^0) = * \implies * = F(d)$

Since the input to C^0 is supplied by the xBook platform, and since $(d, x) \notin \mathbb{P}$, $C^0 \not\rightarrow^* x$.

This is a contradiction. Therefore, $C^i \not\rightarrow^* x$.

By definition, $*$ represents any information (including d).

Therefore, $C^i \not\rightarrow^d x$.

Theorem 2. *Given a set of user policies $P(x) = D \times U$, where the application can pass user $x \in U$ ’s information field $d \in D$ to another user $y \in U$, and assuming that the intended confinement is enforced, the information flow model ensures that user-user access control is enforced in the xBook system. In other words, if $(d, y) \notin P(x)$ then $\forall C_i(x), C_j(y) : C_i(x) \not\rightarrow^d C_j(y)$, where $C_i(x)$ and $C_j(y)$ are components of application instance for user x and y , respectively.*

Proof. Similar to Theorem 1.

7.4 Performance Estimates

xBook does not impose a substantial burden on the performance of the third party applications. With an architectural framework of developing applications, it is difficult to accurately predict the impact of our design on the performance of these applications as perceived by the user. To get a rough estimate of the cost of supporting the xBook design and the overhead involved in our system, we conducted some experiments with our sample applications, measuring latency at the user end and overhead imposed by the mediating design of xBook.

The xBook server side is hosted on a 2.4GHz Pentium 4 machine with 512MB of RAM. The requests are made from Firefox 3.0 browser on a 2.33GHz, 2GB RAM, Pentium Core Duo laptop. Each test was run 10 times and values were averaged. We define user latency as the difference in the time when the request is made at the browser and the time at which the response is received by the browser. Table 2 shows the time required by xBook’s information flow control in comparison to the user’s overall latency. Server processing includes the application’s logic, database access to retrieve required user data, and xBook flow checks, and is independent of the network latency experienced by the application. We instrumented our code to derive the time for performing label checks in the system, and measured overhead as a function of the label checking time over the total latency experienced by the user. Our results show that the overhead introduced by xBook’s label checks is considerably small: about 4% for the horoscope application and 3% for the map utility marking user’s hometown location on Google maps.

On a cluster of commercial servers with much better computational capacity, these values will be even smaller. Although it is not possible to precisely determine the cost of our approach without a large scale experiment, both the details of our design and the results from these

experiments, support the conclusion that xBook design would not substantially increase the latency experienced by users.

8 Discussion

In this section, we discuss the limitations of the application design in xBook and address some of the challenges arising from the new requirements imposed by our design.

Our xBook design imposes no limitations on applications that follow a “pull model”, i.e., xBook would preserve the functionality of applications that only receive data from external entities without passing any private information to these entities. Our horoscope application is an example of such an application: one public component of horoscope pulls horoscope data from `www.tarot.com` and does not pass any of the user’s profile information. Note that the xBook platform does not need to sanitize the request parameters (in both GET and POST requests), as the component making such requests has no user information that can be leaked. Another component, which has access to the user’s birthday information, uses the data to calculate the daily horoscope corresponding to the particular user. This component has no communication with any external entity.

On the other hand, our design might limit some of the applications that require data to be sent to external entities for receiving user-specific information. One typical example is the use of Google APIs to generate maps: it requires a location to be passed to Google before the map is generated. In many cases, we expect these external entities to be larger and well branded entities, such as Google, Yahoo, etc. Such cases could be whitelisted after explicit approval from the user. Note that xBook makes no recommendation about which websites can be trusted, including Google and Yahoo; such trust decisions are made by an individual user from his own knowledge and experiences. Our xBook system can keep track of these approvals across applications for every user, so the users need to approve an interaction only once.

Any social networking application would follow either the pull model or the push model to get data from external entities. In both cases, our platform enforces the applications to make all such interactions explicit and allows the user to make a more informed decision based on the information available. We argue that an application using the pull model would be more acceptable to the users as it requires minimal trust decisions from a user’s perspective. It is possible to transform many of the current social networking applications that use the push model to start using the pull model. We acknowledge that such a transformation would require some changes to the application design, and in some cases, such transformations might not be practical due to large download size of the required data. However, if enough users decide not to use the ap-

plication in view of privacy concerns, it would motivate the developers to consider such a transition.

Our system also suffers from classical covert channels, e.g. timing, memory, process, etc. However, in general these channels have limited bandwidth and viable approaches such as randomizing the time (for example, the delivery time of our message queue discussed in Section 4.3) can further limit their utilities. We plan to study some of these channels as part of our future work.

Scalability of the applications is not a concern in our system: applications hosted on clusters outside xBook would now be hosted on clusters inside the xBook platform. The application developers are already paying for hosting their applications, in most cases to third-parties or cloud owners like Amazon EC2 [2]. Thus, instead of the developers paying to these parties, they would be paying to xBook for the hosting service. xBook, in turn, can out-source the hosting to third-parties, still assuming control of the hosted applications.

We also propose a hybrid model where only the application components that require access to xBook’s private data needs to be hosted at the xBook servers. Other public components can be controlled by the application developers on their own servers. Such an approach is useful for many applications as research has shown that a large number of applications do not use any private data to perform their functionality [19].

9 Related Work

Information flow control at the language level has been well studied [16,27]. Jif is a Java-based programming language that enforces decentralized information flow control within a program, providing finer grained control than xBook [27]. In comparison to these language level techniques that require the applications to be rewritten, the xBook platform provides a simpler interface to the application programmers: they do not need to learn a new language or perform any fine-grained code annotations. Additionally, information flow on a language like javascript with dynamically created source code may not be feasible. Cong et al. [16] presented a technique of writing secure web applications, which generates javascript code on the client side and java code on the server side. However, the applications are still written in the Jif language.

There are other systems [23,36] that have utilized the information flow concept to control data flow at the operating systems (OS) level. Information flows are tracked at low-level OS object types such as threads, processes, etc. xBook works at a much coarser level at the applications, with smallest unit of information being an application component. As a result, run-time information flow in xBook would probably be less expensive as compared to a much finer granularity level used in these systems. In order to make these systems useful for a typical social

networking environment, it would require the systems to be installed at a user's computer because leaks can also happen at the browser, which might not be feasible. In comparison, xBook runs on a typical web server without any changes to the OS environment.

Similar to the ADsafe environment, other safe subsets of programming languages, such as JoeE [20] (for java) and Caja [25] (for javascript), allow third-party applications to provide active content safely and flexibility within the existing web standards. While we used ADsafe for its simplicity and suitability to meet our system needs, we expect that it would be similarly possible to develop xBook using these alternatives.

10 Conclusions

We presented a novel architecture for a social networking framework, called xBook, that substantially improves privacy control in the presence of untrusted third-party application. Our design allows the applications to have access to user data to preserve their functionality, but at the same time preventing them from leaking users' private information.

We developed a working prototype of the system that is available as an application on Facebook [33]. We showed the viability of our system by developing sample applications using the xBook APIs: these applications are similar in functionality to the applications on existing social networks.

Our system shows promise in designing potentially valuable future applications, that would require user data to provide more customized service to the user. The growing popularity of social networks would attract increasing attention from attackers because of the value of user information available in these networks. This user information not only has commercial value, but when combined with some anonymized public data such as medical records, might leak more sensitive information [28, 34]. The current design of social networking applications poses a serious threat to the privacy of individuals that needs to be mitigated; the xBook platform is a major step in protecting user privacy in social networking applications.

Acknowledgement

This material is based upon work supported in part by the NSF under grants no. 0716570 and 0831300 and the Department of Homeland Security under contract no. FA8750-08-2-0141. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF or the Department of Homeland Security. We would also like to thank Monirul Sharif, Roberto Perdisci and the anonymous reviewers for their helpful comments and our shepherd George Danezis for his valuable suggestions.

References

- [1] ADsafe. <http://adsafe.org>. Last accessed Feb. 1, 2009.
- [2] Amazon elastic computing cloud. <http://aws.amazon.com/ec2/>. Last accessed Feb. 1, 2009.
- [3] Daily horoscopes. <http://apps.facebook.com/daily-horoscope>. Last accessed Feb. 1, 2009.
- [4] Facebook developers: Developer terms of service. <http://developers.facebook.com/terms.php>. Last accessed Feb. 1, 2009.
- [5] Facebook's privacy policy. <http://www.facebook.com/policy.php>. Last accessed Feb. 1, 2009.
- [6] Helma javascript web application framework. <http://www.helma.org>.
- [7] Javascript object notation (JSON). <http://www.json.org>. Last accessed Feb. 1, 2009.
- [8] JSLint: The javascript verifier. <http://www.jslint.com>. Last accessed Feb. 1, 2009.
- [9] Map your friends. <http://apps.facebook.com/mapyourfriends>. Last accessed Feb. 1, 2009.
- [10] Opensocial. <http://www.opensocial.org/>. Last accessed Feb. 1, 2009.
- [11] Topfriends. <http://apps.facebook.com/topfriends>. Last accessed Feb. 1, 2009.
- [12] A. Acharya and M. Raje. MAPbox: using parameterized behavior classes to confine untrusted applications. In *Proceedings of the 9th USENIX Security Symposium*, Denver, CO, Aug. 2000.
- [13] L. Backstrom, C. Dwork, and J. Kleinberg. Wherefore art thou r3579x?: Anonymized social networks, hidden patterns, and structural steganography. In *Proceedings of the 16th International Conference on World Wide Web (WWW)*, Banff, Canada, May 2007.
- [14] D. E. Bell and L. J. Lapadula. Secure computer system: Unified exposition and multics interpretation. Technical Report MTR-2997, MITRE Corp., Bedford, MA, Mar. 1976.
- [15] A. Blum, C. Dwork, F. McSherry, and K. Nissim. Practical privacy: the SuLQ framework. In *ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, Baltimore, MD, 2005.
- [16] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. In *Proceedings of the 21st Symposium on Operating Systems Principles (SOSP)*, Stevenson, WA, Oct. 2007.
- [17] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.
- [18] D. Farber. Google to open orkut opensocial developer sandbox tonight, Nov. 2007. <http://blogs.zdnet.com/BTL/?p=6856>. Last accessed Feb. 1, 2009.
- [19] A. Felt and D. Evans. Privacy protection for social networking platforms. In *Web 2.0 Security and Privacy Workshop*, Oakland, CA, May 2008.
- [20] M. Finifter, A. Mettler, N. Sastry, and D. Wagner. Verifiable functional purity in java. In *Proceedings of the ACM Conference on Computer and Communication Security (CCS)*, Alexandria, VA, Oct. 2008.

- [21] S. Hacking. More advertising issues on facebook (updated), 2008. <http://theharmonyguy.com/2008/06/20/more-advertising-issues-on-facebook/>. Last accessed Feb. 1, 2009.
- [22] R. Konrad. Facebook opens to third-party developers, May 2007. <http://www.msnbc.msn.com/id/18899269/>. Last accessed Feb. 1, 2009.
- [23] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *Proceedings of the 21st Symposium on Operating Systems Principles (SOSP)*, Stevenson, WA, Oct. 2007.
- [24] A. Machanavajjhala, D. Kifer, J. Gehrke, and M. Venkatasubramanian. L-diversity: Privacy beyond k-anonymity. *ACM Transactions of Knowledge Discovery from Data*, 1(1):3, 2007.
- [25] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: safe active content in sanitized javascript, Oct. 2007. <http://google-caja.googlecode.com/files/caja-spec-2007-10-11.pdf>.
- [26] E. Mills. Facebook suspends app that permitted peephole, 2008. http://news.cnet.com/8301-10784_3-9977762-7.html. Last accessed Feb. 1, 2009.
- [27] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP)*, Saint-Malo, France, Oct. 1997.
- [28] A. Narayanan and V. Shmatikov. Robust de-anonymization of large sparse datasets. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2008.
- [29] T. Panja. Oxford using Facebook to snoop. <http://www.msnbc.msn.com/id/19813092/>. Last accessed Feb. 1, 2009.
- [30] D. S. Peterson, M. Bishop, and R. Pandey. A flexible containment mechanism for executing untrusted code. In *Proceedings of the 11th USENIX Security Symposium*, San Francisco, CA, Aug. 2002.
- [31] P. Samarati. Protecting respondents' identities in micro-data release. *IEEE Transactions on Knowledge and Data Engineering*, 13(6):1010–1027, 2001.
- [32] D. Sciba. Mayor in myspace photo flap asked to resign. <http://www.katu.com/news/13670287.html>. Last accessed Feb. 1, 2009.
- [33] K. Singh, S. Bhola, and W. Lee. xBook on Facebook. <http://apps.facebook.com/myxbook>. Last accessed Feb. 1, 2009.
- [34] L. Sweeney. Weaving technology and policy together to maintain confidentiality. *Journal of Law, Medicine and Ethics*, 25:98–110, 1997.
- [35] C. Williams. Facebook application hawks your personal opinions for cash, Sept. 2007. http://www.theregister.co.uk/2007/09/12/facebook_compare_people/. Last accessed Feb. 1, 2009.
- [36] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in history. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, Nov. 2006.

Nemesis: Preventing Authentication & Access Control Vulnerabilities in Web Applications

Michael Dalton, Christos Kozyrakis, and Nickolai Zeldovich

Computer Systems Laboratory
Stanford University

CSAIL
MIT

{mwdalton, kozyraki}@stanford.edu nickolai@csail.mit.edu

Abstract

This paper presents *Nemesis*, a novel methodology for mitigating authentication bypass and access control vulnerabilities in existing web applications. *Authentication attacks* occur when a web application authenticates users unsafely, granting access to web clients that lack the appropriate credentials. *Access control attacks* occur when an access control check in the web application is incorrect or missing, allowing users unauthorized access to privileged resources such as databases and files. Such attacks are becoming increasingly common, and have occurred in many high-profile applications, such as IIS [10] and WordPress [31], as well as 14% of surveyed web sites [30]. Nevertheless, none of the currently available tools can fully mitigate these attacks.

Nemesis *automatically determines* when an application safely and correctly authenticates users, by using Dynamic Information Flow Tracking (DIFT) techniques to track the flow of user credentials through the application's language runtime. Nemesis combines authentication information with programmer-supplied access control rules on files and database entries to *automatically ensure* that only properly authenticated users are granted access to any privileged resources or data. A study of seven popular web applications demonstrates that a prototype of Nemesis is effective at mitigating attacks, requires little programmer effort, and imposes minimal runtime overhead. Finally, we show that Nemesis can also improve the precision of existing security tools, such as DIFT analyses for SQL injection prevention, by providing runtime information about user authentication.

1 Introduction

Web applications are becoming increasingly prevalent because they allow users to access their data from any computer and to interact and collaborate with each other. However, exposing these rich interfaces to anyone on the

internet makes web applications an appealing target for attackers who want to gain access to other users' data or resources. Web applications typically address this problem through *access control*, which involves *authenticating* users that want to gain access to the system, and ensuring that a user is properly *authorized* to perform any operation the server executes on her behalf. In theory, this approach should ensure that unauthorized attackers cannot subvert the application.

Unfortunately, experience has shown that many web applications fail to follow these seemingly simple steps, with disastrous results. Each web application typically deploys its own authentication and access control framework. If any flaw exists in the authentication system, an authentication bypass attack may occur, allowing attackers to become authenticated as a valid user without having to present that user's credentials, such as a password. Similarly, a single missing or incomplete access control check can allow unauthorized users to access privileged resources. These attacks can result in the complete compromise of a web application.

Designing a secure authentication and access control system in a web application is difficult. Part of the reason is that the underlying file system and database layers perform operations with the privileges of the web application, rather than with privileges of a specific web application user. As a result, the web application must have the superset of privileges of all of its users. However, much like a Unix *setuid* application, it must explicitly check if the requesting user is authorized to perform each operation that the application performs on her behalf; otherwise, an attacker could exploit the web application's privileges to access unauthorized resources. This approach is ad-hoc and brittle, since these checks must be sprinkled throughout the application code whenever a resource is accessed, spanning code in multiple modules written by different developers over a long period of time. It is hard for developers to keep track of all the security policies that have to be checked. Worse yet, code written for other applications

or third-party libraries with different security assumptions is often reused without considering the security implications. In each case, the result is that it's difficult to ensure the correct checks are always performed.

It is not surprising, then, that authentication and access control vulnerabilities are listed among the top ten vulnerabilities in 2007 [17], and have been discovered in high-profile applications such as IIS [10] and WordPress [31]. In 2008 alone, 168 authentication and access control vulnerabilities were reported [28]. A recent survey of real-world web sites found that over 14% of surveyed sites were vulnerable to an authentication or access control bypass attack [30].

Despite the severity of authentication or authorization bypass attacks, no defensive tools currently exist to automatically detect or prevent them. The difficulty in addressing these attacks stems from the fact that most web applications implement their own user authentication and authorization systems. Hence, it is hard for an automatic tool to ensure that the application properly authenticates all users and only performs operations for which users have the appropriate authorization.

This paper presents *Nemesis*,¹ a security methodology that addresses these problems by *automatically tracking* when user authentication is performed in web applications without relying on the safety or correctness of the existing code. Nemesis can then use this information to *automatically enforce* access control rules and ensure that only authorized web application users can access resources such as files or databases. We can also use the authentication information to improve the precision of other security analyses, such as DIFT-based SQL injection protection, to reduce their false positive rate.

To determine how a web application authenticates users, Nemesis uses Dynamic Information Flow Tracking (DIFT) to track the flow of user credentials, such as a username and password, through the application code. The key insight is that most applications share a similar high-level design, such as storing usernames and passwords in a database table. While the details of the authentication system, such as function names, password hashing algorithms, and session management vary widely, we can nonetheless determine when an application authenticates a user by keeping track of what happens to user credentials at runtime. Once Nemesis detects that a user has provided appropriate credentials, it creates an additional HTTP cookie to track subsequent requests issued by the authenticated user's browser. Our approach does not require the behavior of the application to be modified, and does not require any modifications to the application's existing authentication and access control system. Instead,

¹Nemesis is the Greek goddess of divine indignation and retribution, who punishes excessive pride, evil deeds, undeserved happiness, and the absence of moderation.

Nemesis is designed to secure legacy applications without requiring them to be rewritten.

To prevent unauthorized access in web applications, Nemesis combines user authentication information with authorization policies provided by the application developer or administrator in the form of access control rules for various resources in the application, such as files, directories, and database entries. Nemesis then automatically ensures that these access control rules are enforced at runtime whenever the resource is accessed by an (authenticated) user. Our approach requires only a small amount of work from the programmer to specify these rules—in most applications, less than 100 lines of code. We expect that explicitly specifying access control rules per-resource is less error-prone than having to invoke the access control check each time the resource is accessed, and having to enumerate all possible avenues of attack. Furthermore, in applications that support third-party plugins, these access control rules need only be specified once, and they will automatically apply to code written by all plugin developers.

By allowing programmers to explicitly specify access control policies in their applications, and by tying the authentication information to runtime authorization checks, Nemesis prevents a wide range of authentication and access control vulnerabilities seen in today's applications. The specific contributions of this paper are as follows:

- We present Nemesis, a methodology for inferring authentication and enforcing access control in existing web applications, while requiring minimal annotations from the application developers.
- We demonstrate that Nemesis can be used to prevent authentication and access control vulnerabilities in modern web applications. Furthermore, we show that Nemesis can be used to prevent false positives and improve precision in real-world security tools, such as SQL injection prevention using DIFT.
- We implement a prototype of Nemesis by modifying the PHP interpreter. The prototype is used to collect performance measurements and to evaluate our security claims by preventing authentication and access control attacks on real-world PHP applications.

The remainder of the paper is organized as follows. Section 2 reviews the security architecture of modern web applications, and how it relates to common vulnerabilities and defense mechanisms. We describe our authentication inference algorithm in Section 3, and discuss our access control methodology in Section 4. Our PHP-based prototype is discussed in Section 5. Section 6 presents our experimental results, and Section 7 discusses future work. Finally, Section 8 discusses related work and Section 9 concludes the paper.

2 Web Application Security Architecture

A key problem underlying many security vulnerabilities is that web application code executes with full privileges while handling requests on behalf of users that only have limited privileges, violating the principle of least privilege [11]. Figure 1 provides a simplified view of the security architecture of typical web applications today. As can be seen from the figure, the web application is performing file and database operations on behalf of users using its own credentials, and if attackers can trick the application into performing the wrong operation, they can subvert the application's security. Web application security can thus be viewed as an instance of the confused deputy problem [9]. The rest of this section discusses this architecture and its security ramifications in more detail.

2.1 Authentication Overview

When clients first connect to a typical web application, they supply an application-specific username and password. The web application then performs an authentication check, ensuring that the username and password are valid. Once a user's credentials have been validated, the web application creates a login session for the user. This allows the user to access the web application without having to log in each time a new page is accessed. Login sessions are created either by placing authentication information directly in a cookie that is returned to the user, or by storing authentication information in a session file stored on the server and returning a cookie to the user containing a random, unique session identifier. Thus, a user request is deemed to be authenticated if the request includes a cookie with valid authentication information or session identifier, or if it directly includes a valid username and password.

Once the application establishes a login session for a user, it allows the user to issue requests, such as posting comments on a blog, which might insert a row into a database table, or uploading a picture, which might require a file to be written on the server. However, there is a *semantic gap* between the user authentication mechanism implemented by the web application, and the access control or authorization mechanism implemented by the lower layers, such as a SQL database or the file system. The lower layers in the system usually have no notion of application-level users; instead, database and file operations are usually performed with the privileges and credentials of the web application itself.

Consider the example shown in Figure 1, where the web application writes the file uploaded by user Bob to the local file system and inserts a row into the database to keep track of the file. The file system is not aware of any authentication performed by the web application

or web server, and treats all operations as coming from the web application itself (e.g. running as the Apache user in Unix). Since the web application has access to every user's file, it must perform internal checks to ensure that Bob hasn't tricked it into overwriting some other user's file, or otherwise performing an unauthorized operation. Likewise, database operations are performed using a per-web application database username and password provided by the system administrator, which authenticates the web application as user `webdb` to MySQL. Much like the filesystem layer, MySQL has no knowledge of any authentication performed by the web application, interpreting all actions sent by the web application as coming from the highly-privileged `webdb` user.

2.2 Authentication & Access Control Attacks

The fragile security architecture in today's web applications leads to two common problems, authentication bypass and access control check vulnerabilities.

Authentication bypass attacks occur when an attacker can fool the application into treating his or her requests as coming from an authenticated user, without having to present that user's credentials, such as a password. A typical example of an authentication bypass vulnerability involves storing authentication state in an HTTP cookie without performing any server-side validation to ensure that the client-supplied cookie is valid. For example, many vulnerable web applications store only the username in the client's cookie when creating a new login session. A malicious user can then edit this cookie to change the username to the administrator, obtaining full administrator access. Even this seemingly simple problem affects many applications, including PHP iCalendar [20] and phpFastNews [19], both of which are discussed in more detail in the evaluation section.

Access control check vulnerabilities occur when an access check is missing or incorrectly performed in the application code, allowing an attacker to execute server-side operations that she might not be otherwise authorized to perform. For example, a web application may be compromised by an invalid access control check if an administrative control panel script does not verify that the web client is authenticated as the admin user. A malicious user can then use this script to reset other passwords, or even perform arbitrary SQL queries, depending on the contents of the script. These problems have been found in numerous applications, such as PhpStat [21].

Authentication and access control attacks often result in the same unfettered file and database access as traditional input validation vulnerabilities such as SQL injection and directory traversal. However, authentication and access control bugs are more difficult to detect, because their

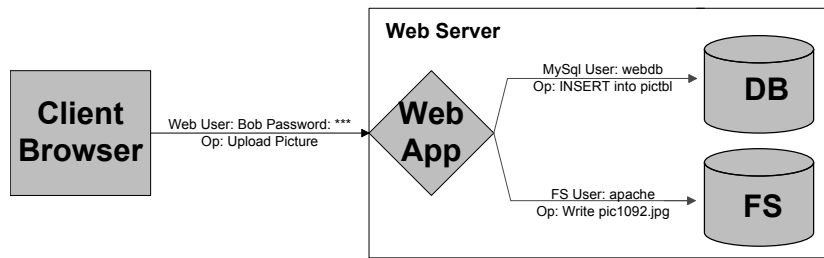


Figure 1: The security architecture of typical web applications. Here, user Bob uploads a picture to a web application, which in turn inserts data into a database and creates a file. The user annotation above each arrow indicates the credentials or privileges used to issue each operation or request.

logic is application-specific, and they do not follow simple patterns that can be detected by simple analysis tools.

2.3 Other Web Application Attacks

Authentication and access control also play an important, but less direct role, in SQL injection [27], command injection, and directory traversal attacks. For example, the PHP code in Figure 2 places user-supplied search parameters into a SQL query without performing any sanitization checks. This can result in a SQL injection vulnerability; a malicious user could exploit it to execute arbitrary SQL statements on the database. The general approach to addressing these attacks is to validate all user input before it is used in any filesystem or database operations, and to disallow users from directly supplying SQL statements. These checks occur throughout the application, and any missing check can lead to a SQL injection or directory traversal vulnerability.

However, these kinds of attacks are effective only because the filesystem and database layers perform all operations with the privilege level of the web application rather than the current authenticated webapp user. If the filesystem and database access of a webapp user were restricted only to the resources that the user should legitimately access, input validation attacks would not be effective as malicious users would not be able to leverage these attacks to access unauthorized resources.

Furthermore, privileged users such as site administrators are often allowed to perform operations that could be interpreted as SQL injection, command injection, or directory traversal attacks. For example, popular PHP web applications such as DeluxeBB and phpMyAdmin allow administrators to execute arbitrary SQL commands. Alternatively, code in Figure 2 could be safe, as long as only administrative users are allowed to issue such search queries. This is the very definition of a SQL injection attack. However, these SQL injection vulnerabilities can only be exploited if the application fails to check that the user is authenticated as the administrator before issuing

the SQL query. Thus, to properly judge whether a SQL injection attack is occurring, the security system must know which user is currently authenticated.

3 Authentication Inference

Web applications often have buggy implementations of authentication and access control, and no two applications have the exact same authentication framework. Rather than try to mandate the use of any particular authentication system, Nemesis prevents authentication and access control vulnerabilities by automatically inferring when a user has been safely authenticated, and then using this authentication information to automatically enforce access control rules on web application users. An overview of Nemesis and how it integrates into a web application software stack is presented in Figure 3. In this section, we describe how Nemesis performs *authentication inference*.

3.1 Shadow Authentication Overview

To prevent authentication bypass attacks, Nemesis must infer when authentication has occurred without depending on the correctness of the application authentication system., which are often buggy or vulnerable. To this end, Nemesis constructs a *shadow authentication system* that works alongside the application's existing authentication framework. In order to infer when user authentication has safely and correctly occurred, Nemesis requires the application developer to provide one annotation—namely, where the application stores user names and their known-good passwords (e.g. in a database table), or what external function it invokes to authenticate users (e.g. using LDAP or OpenID). Aside from this annotation, Nemesis is agnostic to the specific hash function or algorithm used to validate user-supplied credentials.

To determine when a user successfully authenticates, Nemesis uses Dynamic Information Flow Tracking (DIFT). In particular, Nemesis keeps track of two bits

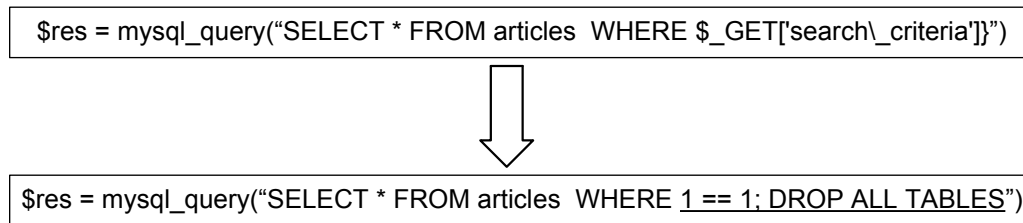


Figure 2: Sample PHP code vulnerable to SQL injection, and the resulting query when a user supplies the underlined, malicious input.

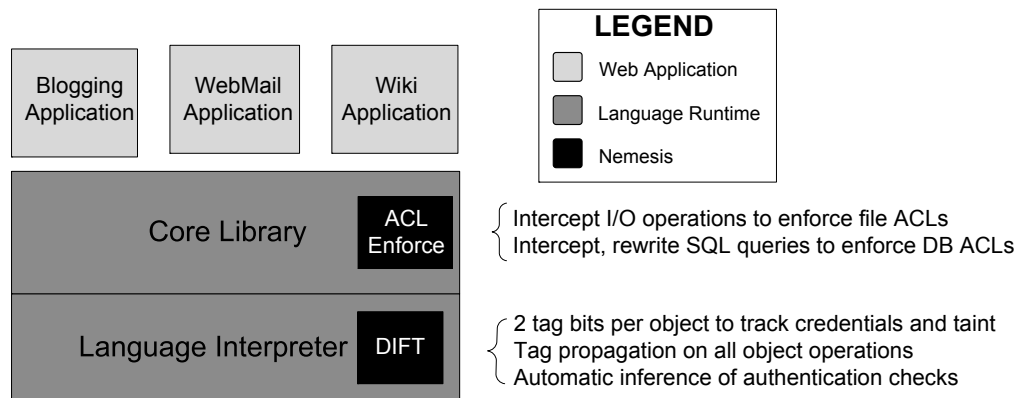


Figure 3: Overview of Nemesis system architecture

of *taint* for each data item in the application—a “credential” taint bit, indicating whether the data item represents a known-good password or other credential, and a “user input” taint bit, indicating whether the data item was supplied by the user as part of the HTTP request. User input includes all values supplied by the untrusted client, such as HTTP request headers, cookies, POST bodies, and URL parameters. Taint bits can be stored either per object (e.g., string), or per byte (e.g., string characters), depending on the needed level of precision and performance.

Nemesis must also track the flow of authentication credentials and user input during runtime code execution. Much like other DIFT systems [8, 15, 16], this is done by performing taint propagation in the language interpreter. Nemesis propagates both taint bits at runtime for all data operations, such as variable assignment, load, store, arithmetic, and string concatenation. The propagation rule we enforce is *union*: a destination operand’s taint bit is set if it was set in any of the source operands. Since Nemesis is concerned with inferring authentication rather than addressing covert channels, implicit taint propagation across control flow is not considered. The rest of this section describes how Nemesis uses these two taint bits to infer when successful authentication has taken place.

3.2 Creating a New Login Session

Web applications commonly authenticate a new user session by retrieving a username and password from a storage location (typically a database) and comparing these credentials to user input. Other applications may use a dedicated login server such as LDAP or Kerberos, and instead defer all authentication to the login server by invoking a special third-party library authentication function. We must infer and detect both of these authentication types.

As mentioned, Nemesis requires the programmer to specify where the application stores user credentials for authentication. Typical applications store password hashes in a database table, in which case the programmer should specify the name of this table and the column names containing the user names and passwords. For applications that defer authentication to an external login server, the programmer must provide Nemesis with the name of the authentication function (such as `ldap_login`), as well as which function arguments represent the username and password, and what value the function returns upon authentication success. In either case, the shadow authentication system uses this information to determine when the web application has safely authenticated a user.

3.2.1 Direct Password Authentication

When an application performs authentication via direct password comparisons, the application must read the username and password from an authentication storage location, and compare them to the user-supplied authentication credentials. Whenever the authentication storage location is read, our shadow authentication system records the username read as the current user under authentication, and sets the “credential” taint bit for the password string. In most web applications, a client can only authenticate as a single user at any given time. If an application allows clients to authenticate as multiple users at the same time, Nemesis would have to be extended to keep track of multiple candidate usernames, as well as multiple “credential” taint bits on all data items. However, we are not aware of a situation in which this occurs in practice.

When data tagged as “user input” is compared to data tagged as “credentials” using string equality or inequality operators, we assume that the application is checking whether a user-supplied password matches the one stored in the local password database. If the two strings are found to be equal, Nemesis records the web client as authenticated for the candidate username. We believe this is an accurate heuristic, because known-good credentials are the only objects in the system with the “credential” taint bit set, and only user input has the “user input” taint bit set. This technique even works when usernames and passwords are supplied via URL parameters (such as “magic URLs” which perform automatic logins in HotCRP) because all values supplied by clients, including URL parameters, are tagged as user input.

Tag bits are propagated across all common operations, allowing Nemesis to support standard password techniques such as cryptographic hashes and salting. Hashing is supported because cryptographic hash functions consist of operations such as array access and arithmetic computations, all of which propagate tag bits from inputs to outputs. Similarly, salting is supported because prepending a salt to a user-supplied password is done via string concatenation, an operation that propagates tag bits from source operands to the destination operand.

This approach allows us to infer user authentication by detecting when a user input string is compared and found to be equal to a password. This avoids any internal knowledge of the application, requiring only that the system administrator correctly specify the storage location of usernames and passwords. A web client will only be authenticated by our shadow authentication system if they know the password, because authentication occurs only when a user-supplied value is equal to a known password. Thus, our approach does not suffer from authentication vulnerabilities, such as allowing a user to log in if a magic URL parameter or cookie value is set.

3.2.2 Deferred Authentication to a Login Server

We use similar logic to detect authentication when using a login server. The web client is assumed to be authenticated if the third-party authentication function is called with a username and password marked as “user input”, and the function returns success. In this case, Nemesis sets the authenticated user to the username passed to this function. Nemesis checks to see if the username and password passed to this function are tainted in order to distinguish between credentials supplied by the web client and credentials supplied internally by the application. For example, phpMyAdmin uses MySQL’s built-in authentication code to both authenticate web clients, and to authenticate itself to the database for internal database queries [23]. Credentials used internally by the application should not be treated as the client’s credentials, and Nemesis ensures this by only accepting credentials that came from the web client. Applications that use single sign-on systems such as OpenID must use deferred authentication, as the third-party authentication server (e.g., OpenID Provider) performs the actual user authentication.

3.3 Resuming a Previous Login Session

As described in Section 2.1, web applications create login sessions by recording pertinent authentication information in cookies. This allows users to authenticate once, and then access the web application without having to authenticate each time a new page is loaded. Applications often write their own custom session management frameworks, and session management code is responsible for many authentication bypass vulnerabilities.

Fortunately, Nemesis does not require any per-application customization for session management. Instead, we use an entirely separate session management framework. When Nemesis infers that user authentication has occurred (as described earlier in this section), a new cookie is created to record the shadow authentication credentials of the current web client. We do not interpret or attempt to validate any other cookies stored and used by the web application for session management. For all intents and purposes, session management in the web application and Nemesis are orthogonal. We refer to the cookie used for Nemesis session management as the *shadow cookie*. When Nemesis is presented with a valid shadow cookie, the current shadow authenticated user is set to the username specified in the cookie.

Shadow authentication cookies contain the shadow authenticated username of the current web user and an HMAC of the username computed using a private key kept on the server. The user cannot edit or change their shadow authentication cookie because the username HMAC will no longer match the username itself, and the user does

not have the key used to compute the HMAC. This cookie is returned to the user, and stored along with any other authentication cookies created by the web application.

Our shadow authentication system detects a user safely resuming a prior login session if a valid shadow cookie is presented. The shadow authentication cookie is verified by recomputing the cookie HMAC based on the username from the cookie. If the recomputed HMAC and the HMAC from the cookie are identical, the user is successfully authenticated by our shadow authentication system. Nemesis distinguishes between shadow cookies from multiple applications running on the same server by using a different HMAC key for each application, and including a hash derived from the application's HMAC key in the name of the cookie.

In practice, when a user resumes a login session, the web application will validate the user's cookies and session file, and then authorize the user to access a privileged resource. When the privileged resource access is attempted, Nemesis will examine the user's shadow authentication credentials and search for valid shadow cookies. If a valid shadow cookie is found and verified to be safe, the user's shadow authentication credentials are updated. Nemesis then performs an access control check on the shadow authentication credentials using the web application ACL.

3.4 Registering a New User

The last way a user may authenticate is to register as a new user. Nemesis infers that new user registration has occurred when a user is inserted into the authentication credential storage location. In practice, this is usually a SQL INSERT statement modifying the user authentication database table. The inserted username must be tainted as "user input", to ensure that this new user addition is occurring on behalf of the web client, and not because the web application needed to add a user for internal usage.

Once the username has been extracted and verified as tainted, the web client is then treated as authenticated for that username, and the appropriate session files and shadow authentication cookies are created. For the common case of a database table, this requires us to parse the SQL query, and determine if the query is an INSERT into the user table or not. If so, we extract the username field from the SQL statement.

3.5 Authentication Bypass Attacks

Shadow authentication information is only updated when the web client supplies valid user credentials, such as a password for a web application user, or when a valid shadow cookie is presented. During authentication bypass attacks, malicious users are authenticated by the web

application without supplying valid credentials. Thus, when one of these attacks occurs, the web application will incorrectly authenticate the malicious web client, but shadow authentication information will not be updated.

While we could detect authentication bypass attacks by trying to discern when shadow authentication information differs from the authenticated state in the web application, this would depend on internal knowledge of each web application's code base. Authentication frameworks are often complex, and each web application typically creates its own framework, possibly spreading the current authentication information among multiple variables and complex data structures.

Instead, we note that the goal of any authentication bypass attack is to use the ill-gotten authentication to obtain unauthorized access to resources. These are exactly the resources that the current shadow authenticated user is not permitted to access. As explained in the next section, we can prevent authentication bypass attacks by detecting when the current shadow authenticated user tries to obtain unauthorized access to a system resource such as a file, directory, or database table.

4 Authorization Enforcement

Both authentication and access control bypass vulnerabilities allow an attacker to perform operations that she would not be otherwise authorized to perform. The previous section described how Nemesis constructs a shadow authentication system to keep track of user authentication information despite application-level bugs. However, the shadow authentication system alone is not enough to prevent these attacks. This section describes how Nemesis mitigates the attacks by connecting its shadow authentication system with an access control system protecting the web application's database and file system.

To control what operations any given web user is allowed to perform, Nemesis allows the application developer to supply access control rules (ACL) for files, directories, and database objects. Nemesis extends the core system library so that each database or file operation performs an ACL check. The ACL check ensures that the current shadow authenticated user is permitted by the web application ACL to execute the operation. This enforcement prevents access control bypass attacks, because an attacker exploiting a missing or invalid access control check to perform a privileged operation will be foiled when Nemesis enforces the supplied ACL. This also mitigates authentication bypass attacks—even if an attacker can bypass the application's authentication system (e.g., due to a missing check in the application code), Nemesis will automatically perform ACL checks against the username provided by the shadow authentication system, which is not subject to authentication bypass attacks.

4.1 Access Control

In any web application, the authentication framework plays a critical role in access control decisions. There are often numerous, complex rules determining which resources (such as files, directories, or database tables, rows, or fields) can be accessed by a particular user. However, existing web applications do not have explicit, codified access control rules. Rather, each application has its own authentication system, and access control checks are interspersed throughout the application.

For example, many web applications have a privileged script used to manage the users of the web application. This script must only be accessed by the web application administrator, as it will likely contain logic to change the password of an arbitrary user and perform other privileged operations. To restrict access appropriately, the beginning of the script will contain an access control check to ensure that unauthorized users cannot access script functionality. This is actually an example of the policy, “only the administrator may access the admin.php script”, or to rephrase such a policy in terms of the resources it affects, “only the administrator may modify the user table in the database”. This policy is often never explicitly stated within the web application, and must instead be inferred from the authorization checks in the web application. Nemesis requires the developer or system administrator to explicitly provide an access control list based on knowledge of the application. Our prototype system and evaluation suggests that, in practice, this requires little programmer effort while providing significant security benefits. Note that a single developer or administrator needs to specify access control rules. Based on these rules, Nemesis will provide security checks for all application users.

4.1.1 File Access

Nemesis allows developers to restrict file or directory access to a particular shadow authenticated user. For example, a news application may only allow the administrator to update the news spool file. We can also restrict the set of valid operations that can be performed: read, write, or append. For directories, read permission is equivalent to listing the contents of the directory, while write permission allows files and subdirectories to be created. File access checks happen before any attempt to open a file or directory. These ACLs could be expressed by listing the files and access modes permitted for each user.

4.1.2 SQL Database Access

Nemesis allows web applications to restrict access to SQL tables. Access control rules specify the user, name of the SQL database table, and the type of access (INSERT, SELECT, DELETE, or UPDATE). For each SQL query,

Nemesis must determine what tables will be accessed by the query, and whether the ACLs permit the user to perform the desired operation on those tables.

In addition to table-level access control, Nemesis also allows restricting access to individual rows in a SQL table, since applications often store data belonging to different users in the same table.

An ACL for a SQL row works by restricting a given SQL table to just those rows that should be accessible to the current user, much like a *view* in SQL terminology. Specifically, the ACL maps SQL table names and access types to an SQL predicate expression involving column names and values that constrain the kinds of rows the current user can access, where the values can be either fixed constants, or the current username from the shadow authentication system, evaluated at runtime. For example, a programmer can ensure that a user can only access their own profile by confining SQL queries on the profile table to those whose *user* column matches the current shadow username.

SELECT ACLs restrict the values returned by a SELECT SQL statement. DELETE and UPDATE query ACLs restrict the values modified by an UPDATE or DELETE statement, respectively. To enforce ACLs for these statements, Nemesis must rewrite the database query to append the field names and values from the ACL to the WHERE condition clause of the query. For example, a query to retrieve a user’s private messages might be “SELECT * FROM messages WHERE recipient=\$current_user”, where \$current_user is supplied by the application’s authentication system. If attackers could fool the application’s authentication system into setting \$current_user to the name of a different user, they might be able to retrieve that user’s messages.

Using Nemesis, the programmer can specify an ACL that only allows SELECTing rows whose sender or recipient column matches the current shadow user. As a result, if user Bob issues the query, Nemesis will transform it into the query “SELECT * FROM messages WHERE recipient=\$current_user AND (sender=Bob or recipient=Bob)”, which mitigates any possible authentication bypass attack.

Finally, INSERT statements do not read or modify existing rows in the database. Thus, access control for INSERT statements is governed solely by the table access control rules described earlier. However, sometimes developers may want to set a particular field to the current shadow authenticated user when a row is inserted into a table. Nemesis accomplishes this by rewriting the INSERT query to replace the value of the designated field with the current shadow authenticated user (or to add an additional field assignment if the designated field was not initialized by the INSERT statement).

Modifying INSERT queries has a number of real-world uses. Many database tables include a field that stores

the username of the user who inserted the field. The administrator can choose to replace the value of this field with the shadow authenticated username, so that authentication flaws do not allow users to spoof the owner of a particular row in the database. For example, in the PHP forum application DeluxeBB, we can override the author name field in the table of database posts with the shadow authenticated username. This prevents malicious clients from spoofing the author when posting messages, which can occur if an authentication flaw allows attackers to authenticate as arbitrary users.

4.2 Enhancing Access Control with DIFT

Web applications often perform actions which are not authorized for the currently authenticated user. For example, in the PHP image gallery Linpha, users may inform the web application that they have lost their password. At this point, the web client is unauthenticated (as they have no valid password), but the web application changes the user's password to a random value, and e-mails the new password to the user's e-mail account. While one user should not generally be allowed to change the password of a different user, doing so is safe in this case because the application generates a fresh password not known to the requesting user, and only sends it via email to the owner's address.

One heuristic that helps us distinguish these two cases in practice is the taint status of the newly-supplied password. Clearly it would be a bad idea to allow an unauthenticated user to supply the new password for a different user's account, and such password values would have the "user input" taint under Nemesis. At the same time, our experience suggests that internally-generated passwords, which do not have the "user input" taint, correspond to password reset operations, and would be safe to allow.

To support this heuristic, we add one final parameter to all of the above access control rules: taint status. An ACL entry may specify, in addition to its other parameters, taint restrictions for the file contents or database query. For example, an ACL for Linpha allows the application to update the password field of the user table regardless of the authentication status, as long as the query is untainted. If the query is tainted, however, the ACL only allows updates to the row corresponding to the currently authenticated user.

4.3 Protecting Authentication Credentials

Additionally, there is one security rule that does not easily fit into our access control model, yet can be protected via DIFT. When a web client is authenticating to the web application, the application must read user credentials such as a password and use those credentials to authenticate

the client. However, unauthenticated clients do not have permission to see passwords. A safe web application will ensure that these values are never leaked to the client. To prevent an information leak bug in the web application from resulting in password disclosure, Nemesis forbids any object that has the authentication credential DIFT tag bit set from being returned in any HTTP response. In our prototype, this rule has resulted in no false positives in practice. Nevertheless, we can easily modify this rule to allow passwords for a particular user to be returned in a HTTP response once the client is authenticated for that user. For example, this situation could arise if a secure e-mail service used the user's password to decrypt e-mails, causing any displayed emails to be tagged with the password bit.

5 Prototype Implementation

We have implemented a proof-of-concept prototype of Nemesis by modifying the PHP interpreter. PHP is one of the most popular languages for web application development. However, the overall approach is not tied to PHP by design, and could be implemented for any other popular web application programming language. Our prototype is based on an existing DIFT PHP tainting project [29]. We extend this work to support authentication inference and authorization enforcement.

5.1 Tag Management

PHP is a dynamically typed language. Internally, all values in PHP are instances of a single type of structure known as a *zval*, which is stored as a tagged union. Integers, booleans, and strings are all instances of the *zval* struct. Aggregate data types such as arrays serve as hash tables mapping index values to *zvals*. Symbol tables are hash tables mapping variable names to *zvals*.

Our prototype stores taint information at the granularity of a *zval* object, which can be implemented without storage overhead in the PHP interpreter. Due to alignment restrictions enforced by GCC, the *zval* structure has a few unused bits, which is sufficient for us to store the two taint bits required by Nemesis.

By keeping track of taint at the object level, Nemesis assumes that the application will not combine different kinds of tagged credentials in the same object (e.g. by concatenating passwords from two different users together, or combining untrusted and authentication-based input into a single string). While we have found this assumption to hold in all encountered applications, a byte granularity tainting approach could be used to avoid this limitation if needed, and prior work has shown it practical to implement byte-level tainting in PHP [16]. When multiple objects are combined in our prototype, the result's taint

bits are the union of the taint bits on all inputs. This works well for combining tainted and untainted data, such as concatenating an untainted salt with a tainted password (with the result being tainted), but can produce imprecise results when concatenating objects with two different classes of taint.

User input and password taint is propagated across all standard PHP operations, such as variable assignment, arithmetic, and string concatenation. Any value with password taint is forbidden from being returned to the user via echo, printf, or other output statements.

5.2 Tag Initialization

Any input from URL parameters (GET, PUT, etc), as well as from any cookies, is automatically tainted with the 'user input' taint bit. Currently, password taint initialization is done by manually inserting the taint initialization function call as soon as the password enters the system (e.g., from a database) as we have not yet implemented a full policy language for automated credential tainting. For a few of our experiments in Section 6 (phpFastNews, PHP iCalendar, Bilboblog), the admin password was stored in a configuration php script that was included by the application scripts at runtime. In this case, we instrumented the configuration script to set the password bit of the admin password variable in the script.

At the same time as we initialize the password taint, we also set a global variable to store the candidate username associated with the password, to keep track of the current username being authenticated. If authentication succeeds, the shadow authentication system uses this candidate username to set the global variable that stores the shadow authenticated user, as well as to initialize the shadow cookie. If a client starts authenticating a second time as a different user, the candidate username is reset to the new value, but the authenticated username is not affected until authentication succeeds.

Additionally, due to an implementation artifact in the PHP `setcookie()` function, we also record shadow authentication in the PHP built-in session when appropriate. This is because PHP forbids new cookies to be added to the HTTP response once the application has placed part of the HTML body in the response output buffer. In an application that uses PHP sessions, the cookie only stores the session ID and all authentication information is stored in session files on the server. These applications may output part of the HTML body before authentication is complete. We correctly handle this case by storing shadow authentication credentials in the server session file if the application has begun a PHP session. When validating and recovering shadow cookies for authentication purposes, we also check the session file associated with the current user for shadow authentication credentials.

This approach relies on PHP safely storing session files, but as session files are stored on the server in a temporary directory, this is a reasonable assumption.

5.3 Authentication Checks

When checking the authentication status of a user, we first check the global variable that indicates the current shadow authenticated user. This variable is set if the user has just begun a new session and been directly authenticated via password comparison or deferred authentication to a login server. If this variable is not set, we check to see if shadow authentication information is stored in the current session file (if any). Finally, we check to see if the user has presented a shadow authentication cookie, and if so we validate the cookie and extract the authentication credentials. If none of these checks succeeds, the user is treated as unauthenticated.

5.4 Password Comparison Authentication Inference

Authentication inference for password comparisons is performed by modifying the PHP interpreter's string comparison equality and inequality operators. When one of these string comparisons executes, we perform a check to see if the two string operands were determined to be equal. If the strings were equal, we then check their tags, and if one string has only the authentication credential tag bit set, and the other string has only the user input tag bit set, then we determine that a successful shadow authentication has occurred. In all of our experiments, only PhpMyAdmin used a form of authentication that did not rely on password string comparison, and our handling of this case is discussed in Section 6.

5.5 Access Control Checks

We perform access control checks for files by checking the current authenticated user against a list of accessible files (and file modes) on each file access. Similarly, we restrict SQL queries by checking if the currently authenticated user is authorized to access the table, and by appending additional WHERE clause predicates to scope the effect of the query to rows allowed for the current user.

Due to time constraints, we manually inserted these checks into applications based on the ACL needed by the application. ACLs that placed constraints on field values of a database row required simple query modifications to test if the field value met the constraints in the ACL.

In a full-fledged design, the SQL queries should be parsed, analyzed for the appropriate information, and rewritten if needed to enforce additional security guarantees (e.g., restrict rows modified to be only those cre-

ated by the current authenticated user). Depending on the database used, query rewriting may also be partially or totally implemented using database views and triggers [18, 26].

5.6 SQL Injection

Shadow authentication is necessary to prevent authentication bypass attacks and enforce our ACL rules. However, it can also be used to prevent false positives in DIFT SQL injection protection analyses. The most robust form of SQL injection protection [27] forbids tainted keywords or operators, and enforces the rule that tainted data may never change the parse tree of a query.

Our current approach does not support byte granularity taint, and thus we must approximate this analysis. We introduce a third taint bit in the *zval* which we use to denote user input that may be interpreted as a SQL keyword or operator. We scan all user input at startup (GET, POST, COOKIE superglobal arrays, etc) and set this bit only for those user input values that contain a SQL keyword or operator. SQL quoting functions, such as `mysql_real_escape_string()`, clear this tag bit. Any attempt to execute a SQL query with the unsafe SQL tag bit set is reported as a SQL injection attack.

We use this SQL injection policy to confirm that DIFT SQL Injection has false positives and real-world web applications. This is because DIFT treats all user input as untrusted, but some web applications allow privileged users such as the admin to submit full SQL queries. As discussed in Section 6, we eliminate all encountered false positives using authentication policies which restrict SQL injection protection to users that are not shadow authenticated as the admin user. We have confirmed that all of these false positives are due to a lack of authentication information, and not due to any approximations made in our SQL injection protection implementation.

6 Experimental Results

To validate Nemesis, we used our prototype to protect a wide range of vulnerable real-world PHP applications from authentication and access control bypass attacks. A summary of the applications and their vulnerabilities is given in Table 1, along with the lines of code that were added or modified in order to protect them.

For each application, we had to specify where the application stores its username and password database, or what function it invokes to authenticate users. This step is quite simple for all applications, and the “authentication inference” column indicates the amount of code we had to add to each application to specify the table used to store known-good passwords, and to taint the passwords with the “credential” taint bit.

We also specified ACLs on files and database tables to protect them from unauthorized accesses; the number of access control rules for each application is shown in the table. As explained in Section 5, we currently enforce ACLs via explicitly inserted checks, which slightly increases the lines of code needed to implement the check (shown in the table as well). As we develop a full MySQL parser and query rewriter, we expect the lines of code needed for these checks to drop further.

We validated our rules by using each web application extensively to ensure there are no false positives, and then verifying that our rules prevented real-world attacks that have been found in these applications in the past. We also verified that our shadow authentication information is able to prevent false positives in DIFT SQL injection analyses for both the DeluxeBB and phpMyAdmin applications.

6.1 PHP iCalendar

PHP iCalendar is a PHP web application for presenting calendar information to users. The webapp administrator is authenticated using a configuration file that stores the admin username and password. Our ACL for PHP iCalendar allows users read access to various template files, language files, and all of the calendars. In addition, caches containing parsed calendars can be read or written by any user. The admin user is able to write, create, and delete calendar files, as well as read any uploaded calendars from the uploads directory. We added 8 authorization checks to enforce our ACL for PHP iCalendar.

An authentication bypass vulnerability occurs in PHP iCalendar because a script in the admin subdirectory incorrectly validates a login cookie when resuming a session [20]. This vulnerability allows a malicious user to forge a cookie that will cause her to be authenticated as the admin user.

Using Nemesis, when an attacker attempts to exploit the authentication bypass attack, she will find that her shadow authentication username is not affected by the attack. This is because shadow authentication uses its own secure form of cookie authentication, and stores its credentials separately from the rest of the web application. When the attacker attempts to use the admin scripts to perform any actions that require admin access, such as deleting a calendar, a security violation is reported because the shadow authentication username will not be ‘admin’, and the ACL will prevent that username from performing administrative operations.

6.2 Phpstat

Phpstat is an application for presenting a database of IM statistics to users, such as summaries and logs of their IM

Program	LoC in program	LoC for auth inference	Number of ACL checks	LoC for ACL checks	Vulnerability prevented
PHP iCalendar	13500	3	8	22	Authentication bypass
phpStat (IM Stats)	12700	3	10	17	Missing access check
Bilboblog	2000	3	4	11	Invalid access check
phpFastNews	500	5	2	17	Authentication bypass
Linpha Image Gallery	50000	15	17	49	Authentication bypass
DeluxeBB Web Forum	22000	6	82	143	Missing access check

Table 1: Applications used to evaluate Nemesis.

conversations. Phpstat stores its authentication credentials in a database table.

The access control list for PhpStat allows users to read and write various cache files, as well as read the statistics database tables. Users may also read profile information about any other user, but the value of the password field may never be sent back to the Web client. The administrative user is also allowed to create users by inserting into or updating the users table, as well as all of the various statistics tables. We added 10 authorization checks to enforce our ACL for PhpStat.

A security vulnerability exists in PhpStat because an installation script will reset the administrator password if a particular URL parameter is given. This behavior occurs without any access control checks, allowing any user to reset the admin password to a user-specified value [21]. Successful exploitation grants the attacker full administrative privileges to the Phpstat. Using Nemesis, when this attack occurs, the attacker will not be shadow authenticated as the admin, and any attempts to execute a SQL query that changes the password of the administrator are denied by our ACL rules. Only users shadow authenticated as the admin may change passwords.

6.3 Bilboblog

Bilboblog is a simple PHP blogging application that authenticates its administrator using a username and password from a configuration file.

Our ACL for bilboblog permits all users to read and write blog caching directories, and read any posted articles from the article database table. Only the administrator is allowed to modify or insert new entries into the articles database table. Bilboblog has an invalid access control check vulnerability because one of its scripts, if directly accessed, uses uninitialized variables to authenticate the admin user [3]. We added 4 access control checks to enforce our ACL for bilboblog.

In PHP, if the `register_globals` option is set, uninitialized variables may be initialized at startup by user-supplied URL parameters [22]. This allows a malicious user to supply the administrator username and password

that the login will be authenticated against. The attacker may simply choose a username and password, access the login script with these credentials encoded as URL parameters, and then input the same username and password when prompted by the script. This attack grants the attacker full administrative access to Bilboblog.

This kind of attack does not affect shadow authentication. A user is shadow authenticated only if their input is compared against a valid password. This attack instead compares user input against a URL parameter. URL parameters do not have the password bit set – only passwords read from the configuration do. Thus, no shadow authentication occurs when this attack succeeds. If an attacker exploits this vulnerability on a system protected by our prototype, she will find herself unable to perform any privileged actions as the admin user. Any attempt to update, delete, or modify an article will be prevented by our prototype, as the current user will not be shadow authenticated as the administrator.

6.4 phpFastNews

PhpFastNews is a PHP application for displaying news stories. It performs authentication via a configuration file with username and password information. This web application displays a news spool to users. Our ACL for phpFastNews allows users to read the news spool, and restricts write access to the administrator. We added 2 access control checks to enforce our ACL for phpFastNews.

An authentication bypass vulnerability occurs in phpFastNews due to insecure cookie validation [19], much like in PHP iCalendar. If a particular cookie value is set, the user is automatically authenticated as the administrator without supplying the administrator's password. All the attacker must do is forge the appropriate cookie, and full admin access is granted.

Using Nemesis, when the authentication bypass attack occurs, our prototype will prevent any attempt to perform administrator-restricted actions such as updating the news spool because the current user is not shadow authenticated as the admin.

6.5 Linpha

Linpha is a PHP web gallery application, used to display directories of images to web users. It authenticates its users via a database table.

Our ACL for Linpha allows users to read files from the images directory, read and write files in the temporary and cache directories, and insert entries into the thumbnails table. Users may also read from the various settings, group, and configuration tables. The administrator may update or insert into the users table, as well as the settings, groups, and categories tables. Dealing with access by non-admin users to the user table is the most complex part of the Linpha ACL, and is our first example of a database row ACL. Any user may read from the user table, with the usual restriction that passwords may never be output to the Web client via echo, print, or related commands.

Users may also update entries in the user table. Updating the password field must be restricted so that a malicious user cannot update the other passwords. This safety restriction can be enforced by ensuring that only user table rows that have a username field equal to the current shadow authenticated user can be modified. The exception to this rule is when the new password is untainted. This can occur only when the web application has internally generated the new user password without using user input. We allow these queries even when they affect the password of a user that is not the current shadow authenticated user because they are used for lost password recovery.

In Linpha, users may lose their password, in which case Linpha resets their password to an internally generated value, and e-mails this password to the user. This causes an arbitrary user's password to be changed on the behalf of a user who isn't even authenticated. However, we can distinguish this safe and reasonable behavior from an attack by a user attempting to change another user's password by examining the taint of the new password value in the SQL query. Thus, we allow non-admin users to update the password field of the user table if the password query is untainted, or if the username of the modified row is equal to the current shadow authenticated user. Overall, we added 17 authorization checks to enforce all of our ACLs for Linpha.

Linpha also has an authentication bypass vulnerability because one of its scripts has a SQL injection vulnerability in the SQL query used to validate login information from user cookies [13]. Successful exploitation of this vulnerability grants the attacker full administrative access to Linpha. For this experiment, we disabled SQL injection protection provided by the tainting framework we used to implement the Nemesis prototype [29], to allow the user to submit a malicious SQL query in order to bypass authentication entirely.

Using Nemesis, once a user has exploited this authentication bypass vulnerability, she may access the various administration scripts. However, any attempt to actually use these scripts to perform activities that are reserved for the admin user will fail, because the current shadow authenticated user will not be set to admin, and our ACLs will correspondingly deny any admin-restricted actions.

6.6 DeluxeBB

DeluxeBB is a PHP web forum application that supports a wide range of features, such as an administration console, multiple forums, and private message communication between forum users. Authentication is performed using a table from a MySQL database.

DeluxeBB has the most intricate ACL of any application in our experiments. All users in DeluxeBB may read and write files in the attachment directory, and the admin user may also write to system log files. Non-admin users in DeluxeBB may read the various configuration and settings tables. Admin users can also write these tables, as well as perform unrestricted modifications to the user table. DeluxeBB treats user table updates and lost password modifications in the same manner as Linpha, and we use the equivalent ACL to protect the user table from non-admin modifications and updates.

DeluxeBB allows unauthenticated users to register via a form, and thus unauthenticated users are allowed to perform inserts into the user table. As described in Section 3.4, inserting a user into the user table results in shadow authentication with the credentials of the inserted user.

The novel and interesting part of the ACLs for DeluxeBB are the treatment of posts, thread creation, and private messages. All inserts into the post, thread creation, or private message tables are rewritten to use the shadow authenticated user as the value for the author field (or the sender field, in the case of a private message). The only exception is when a query is totally untainted. For example, when a new user registers, a welcome message is sent from a fake system mailer user. As this query is totally untainted, we allow it to be inserted into the private message table, despite the fact that the identity of the sender is clearly forged. We added fields to the post and thread tables to store the username of the current shadow authenticated user, as these tables did not directly store the author's username. We then explicitly instrumented all SQL INSERT statements into these tables to append this information accordingly.

Any user may read from the thread or post databases. However, our ACL rules further constrain reads from the private message database. A row may only be read from the private message database if the 'from' or 'to' fields of the row are equal to the current shadow authenticated user.

We manually instrumented all SELECT queries from the private message table to add this condition to the WHERE clause of the query. In total, we modified 16 SQL queries to enforce both our private message protection and our INSERT rules to prevent spoofing messages, threads, and posts. We also inserted 82 authorization checks to enforce the rest of the ACL.

A vulnerability exists in the private message script of this application [6]. This script incorrectly validates cookies, missing a vital authentication check. This allows an attacker to forge a cookie and be treated as an arbitrary web application user by the script. Successful exploitation of this vulnerability gives an attacker the ability to access any user's private messages.

Using Nemesis, when this attack is exploited, the attacker can fool the private message script into thinking he is an arbitrary user due to a missing access control check. The shadow authentication for the attack still has the last safe, correct authenticated username, and is not affected by the attack. Thus, the attacker is unable to access any unauthorized messages, because our ACL rules only allow a user to retrieve messages from the private message table when the sender or recipient field of the message is equal to the current shadow authenticated user. Similarly, the attacker cannot abuse the private message script to forge messages, as our ACLs constrain any messages inserted into the private message table to have the sender field set to the current shadow authenticated username.

DeluxeBB allows admin users to execute arbitrary SQL queries. We verified that this results in false positives in existing DIFT SQL injection protection analyses. After adding an ACL allowing SQL injection for web clients shadow authenticated as the admin user, all SQL injection false positives were eliminated.

6.7 PhpMyAdmin

PhpMyAdmin is a popular web application used to remotely administer and manage MySQL databases. This application does not build its own authentication system; instead, it checks usernames and passwords against MySQL's own user database. A web client is validated only if the underlying MySQL database accepts their username and password.

We treat the MySQL database connection function as a third-party authentication function as detailed in Section 3.2.2. We instrumented the call to the MySQL database connection function to perform shadow authentication, authenticating a user if the username and password supplied to the database are both tainted, and if the login was successful.

The ACL for phpMyAdmin is very different from other web applications, as phpMyAdmin is intended to provide an authenticated user with unrestricted access to the un-

derlying database. The only ACL we include is a rule allowing authenticated users to submit full SQL database queries. We implemented this by modifying our SQL injection protection policy defined in Section 5.6 to treat tainted SQL operators in user input as unsafe only when the current user was unauthenticated. Without this policy, any attempt to submit a query as an authenticated user results in a false positive in the DIFT SQL injection protection policy. We confirmed that adding this ACL removes all observed SQL injection false positives, while still preventing unauthenticated users from submitting SQL queries.

6.8 Performance

We also performed performance tests on our prototype implementation, measuring overhead against an unmodified version of PHP. We used the bench.php microbenchmark distributed with PHP, where our overhead was 2.9% compared to unmodified PHP. This is on par with prior results reported by object-level PHP tainting projects [29]. However, bench.php is a microbenchmark which performs CPU-intensive operations. Web applications often are network or I/O-bound, reducing the real-world performance impact of our information flow tracking and access checks.

To measure performance overhead of our prototype for web applications, we used the Rice University Bidding System (RUBiS) [24]. RUBiS is a web application benchmark suite, and has a PHP implementation that is approximately 2,100 lines of code. Our ACL for RUBiS prevents users from impersonating another user when placing bids, purchasing an item, or posting a bid comment. Three access checks were added to enforce this ACL. We compared latency and throughput for our prototype and an unmodified PHP, and found no discernible performance overhead.

7 Future Work

There is much opportunity for further research in preventing authentication bypass attacks. A fully developed, robust policy language for expressing access control in web applications must be developed. This will require support for SQL query rewriting, which can be implemented by developing a full SQL query parser or by using database table views and triggers similar to the approach described in [18].

Additionally, all of our ACL rules are derived from real-world behavior observed when using the web application. While we currently generate such ACLs manually, it should be possible to create candidate ACLs automatically. A log of all database and file operations, as well as the current shadow authenticated user at the time of the

operation, would be recorded. Using statistical techniques such as machine learning [14], this log could be analyzed and access control files generated based on application behavior. This would allow system administrators to automatically generate candidate ACL lists for their web applications without a precise understanding of the access control rules used internally by the webapp.

8 Related Work

Preventing web authentication and authorization vulnerabilities is a relatively new area of research. The only other work in this area of which we are aware is the CLAMP research project [18]. CLAMP prevents authorization vulnerabilities in web applications by migrating the user authentication module of a web application into a separate, trusted virtual machine (VM). Each new login session forks a new, untrusted session VM and forwards any authentication credentials to the authentication VM. Authorization vulnerabilities are prevented by a trusted query restricter VM which interposes on all session VM database accesses, examining queries to enforce the appropriate ACLs using the username supplied by the authentication VM.

In contrast to Nemesis, CLAMP does not prevent authentication vulnerabilities as the application authentication code is part of the trusted user authentication VM. CLAMP also cannot support access control policies that require taint information as it does not use DIFT. Furthermore, CLAMP requires a new VM to be forked when a user session is created, and experimental results show that one CPU core could only fork two CLAMP VMs per second. This causes significant performance degradation for throughput-driven, multi-user web applications that have many simultaneous user sessions.

Researchers have extensively explored the use of DIFT to prevent security vulnerabilities. Robust defenses against SQL injection [15, 27], cross-site scripting [25], buffer overflows [5] and other attacks have all been proposed. DIFT has been used to prevent security vulnerabilities in most popular web programming languages, including Java [8], PHP [16], and even C [15]. This paper shows how DIFT techniques can be used to address authentication and access control vulnerabilities. Furthermore, DIFT-based solutions to attacks such as SQL injection have false positives in the real-world due to a lack of authentication information. Nemesis avoids such false positives by incorporating authentication and authorization information at runtime.

Much work has also been done in the field of secure web application design. Information-flow aware programming language extensions such as Sif [4] have been developed to prevent information leaks and security vulnerabilities in web application programs using the language

and compiler. Unfortunately these approaches typically require legacy applications to be rewritten in the new, secure programming language.

Some web application frameworks provide common authentication or authorization code libraries [1, 2, 7]. The use of these authentication libraries is optional, and many application developers choose to partially or entirely implement their own authentication and authorization systems. Many design decisions, such as how users are registered, how lost passwords are recovered, or what rules govern access control to particular database tables are application-specific. Moreover, these frameworks do not connect the user authentication mechanisms with the access control checks in the underlying database or file system. As a result, the application programmer must still apply access checks at every relevant filesystem and database operation, and even a single mistake can compromise the security of the application.

Operating systems such as HiStar [32] and Flume [12] provide process-granularity information flow control support. One of the key advantages of these systems is that they give applications the flexibility to define their own security policies (in terms of information flow categories), which are then enforced by the underlying kernel. A web application written for HiStar or Flume can implement its user authentication logic in terms of the kernel's information flow categories. This allows the OS kernel to then ensure that one web user cannot access data belonging to a different web user, even though the OS kernel doesn't know how to authenticate a web user on its own. Our system provides two distinct advantages over HiStar and Flume. First, we can mitigate vulnerabilities in existing web applications, without requiring the application to be re-designed from scratch for security. Second, by providing sub-process-level information flow tracking and expressive access control checks instead of labels, we allow programmers to specify precise security policies in a small amount of code.

9 Conclusion

This paper presented Nemesis, a novel methodology for preventing authentication and access control bypass attacks in web applications. Nemesis uses Dynamic Information Flow Tracking to automatically detect when application-specific users are authenticated, and constructs a *shadow authentication system* to track user authentication state through an additional HTTP cookie.

Programmers can specify access control lists for resources such as files or database entries in terms of application-specific users, and Nemesis automatically enforces these ACLs at runtime. By providing a shadow authentication system and tightly coupling the authentication system to authorization checks, Nemesis prevents

a wide range of authentication and access control bypass attacks found in today's web applications. Nemesis can also be used to improve precision in other security tools, such as those that find SQL injection bugs, by avoiding false positives for properly authenticated requests.

We implemented a prototype of Nemesis in the PHP interpreter, and evaluated its security by protecting seven real-world applications. Our prototype stopped all known authentication and access control bypass attacks in these applications, while requiring only a small amount of work from the application developer, and introducing no false positives. For most applications we evaluated, the programmer had to write less than 100 lines of code to avoid authentication and access control vulnerabilities. We also measured performance overheads using PHP benchmarks, and found that our impact on web application performance was negligible.

Acknowledgments

We would like to thank the anonymous reviewers for their feedback. Michael Dalton was supported by a Sequoia Capital Stanford Graduate Fellowship. This work was supported NSF Awards number 0546060 and 0701607.

References

- [1] Turbogears documentation: Identity management. <http://docs.turbogears.org/1.0/Identity>.
- [2] Zope security. <http://www.zope.org/Documentation/Books/ZDG/current/Security.stx>.
- [3] BilboBlog admin/index.php Authentication Bypass Vulnerability. <http://www.securityfocus.com/bid/30225>, 2008.
- [4] S. Chong, K. Vikram, and A. C. Myers. Sif: Enforcing confidentiality and integrity in web applications. In *Proceedings of the 16th Annual USENIX Security Symposium*, 2007.
- [5] M. Dalton, H. Kannan, and C. Kozyrakis. Real-World Buffer Overflow Protection for Userspace and Kernelspace. In *Proceedings of the 17th Annual USENIX Security Symposium*, 2008.
- [6] DeluxeBB PM.PHP Unauthorized Access Vulnerability. <http://www.securityfocus.com/bid/19418>, 2006.
- [7] Django Software Foundation. User authentication in Django. <http://docs.djangoproject.com/en/dev/topics/auth/>.
- [8] V. Haldar, D. Chandra, and M. Franz. Dynamic taint propagation for java. *Annual Computer Security Applications Conference*, 2005.
- [9] N. Hardy. The Confused Deputy: (or why capabilities might have been invented). *SIGOPS Operating System Review*, 1988.
- [10] Microsoft Internet Information Server Hit Highlighting Authentication Bypass Vulnerability. <http://www.securityfocus.com/bid/24105>, 2007.
- [11] M. Krohn, P. Efstathopoulos, C. Frey, F. Kaashoek, E. Kohler, D. Mazières, R. Morris, M. Osborne, S. VanDeBogart, and D. Ziegler. Make Least Privilege a Right (Not a Privilege). In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*, 2005.
- [12] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles*, 2007.
- [13] Linpha User Authentication Bypass Vulnerability. <http://secunia.com/advisories/12189>, 2004.
- [14] E. Martin and T. Xie. Inferring access-control policy properties via machine learning. In *Proc. 7th IEEE Workshop on Policies for Distributed Systems and Networks*, 2006.
- [15] S. Nanda, L.-C. Lam, and T. Chiueh. Dynamic multi-process information flow tracking for web application security. In *Proceedings of the 8th International Conference on Middleware*, 2007.
- [16] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically Hardening Web Applications using Precise Tainting. In *Proceedings of the 20th IFIP Intl. Information Security Conference*, 2005.
- [17] Top 10 2007 - Broken Authentication and Session Management. http://www.owasp.org/index.php/Top_10_2007-A7, 2007.
- [18] B. Parno, J. M. McCune, D. Wendlandt, D. G. Andersen, and A. Perrig. CLAMP: Practical Prevention of Large-Scale Data Leaks. In *Proceedings of the 2009 IEEE Symposium on Security and Privacy*, May 2009.
- [19] phpFastNews Cookie Authentication Bypass Vulnerability. <http://www.securityfocus.com/bid/31811>, 2008.
- [20] PHP iCalendar Cookie Authentication Bypass Vulnerability. <http://www.securityfocus.com/bid/31320>, 2008.
- [21] Php Stat Vulnerability Discovery. http://www.soulblack.com.ar/repo/papers/advisory/PhpStat_advisory.txt, 2005.
- [22] PHP: Using Register Globals. http://us2.php.net/register_globals.
- [23] PhpMyAdmin control user. <http://wiki.cihar.com/pma/controluser>.
- [24] Rice University Bidding System. <http://rubis.objectweb.org>, 2009.
- [25] P. Saxena, D. Song, and Y. Nadjji. Document structure integrity: A robust basis for cross-site scripting defense. *Network and Distributed Systems Security Symposium*, 2009.
- [26] S. R. Shariq, A. Mendelzon, S. Sudarshan, and P. Roy. Extending Query Rewriting Techniques for Fine-Grained Access Control. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2004.
- [27] Z. Su and G. Wassermann. The Essence of Command Injection Attacks in Web Applications. In *Proceedings of the 33rd Symposium on Principles of Programming Languages*, 2006.
- [28] The MITRE Corporation. Common vulnerabilities and exposures (CVE) database. <http://cve.mitre.org/data/downloads/>.
- [29] W. Venema. Taint support for php. <http://wiki.php.net/rfc/taint>, 2008.
- [30] Web Application Security Consortium. 2007 web application security statistics. http://www.webappsec.org/projects/statistics/wasc_wass_2007.pdf.
- [31] WordPress Cookie Integrity Protection Unauthorized Access Vulnerability. <http://www.securityfocus.com/bid/28935>, 2008.
- [32] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in histar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, 2006.

Static Enforcement of Web Application Integrity Through Strong Typing

William Robertson
wkr@cs.ucsb.edu
Computer Security Group
UC Santa Barbara

Giovanni Vigna
vigna@cs.ucsb.edu
Computer Security Group
UC Santa Barbara

Abstract

Security vulnerabilities continue to plague web applications, allowing attackers to access sensitive data and co-opt legitimate web sites as a hosting ground for malware. Accordingly, researchers have focused on various approaches to detecting and preventing common classes of security vulnerabilities in web applications, including anomaly-based detection mechanisms, static and dynamic analyses of server-side web application code, and client-side security policy enforcement.

This paper presents a different approach to web application security. In this work, we present a web application framework that leverages existing work on strong type systems to statically enforce a separation between the *structure* and *content* of both web documents and database queries generated by a web application, and show how this approach can automatically prevent the introduction of both server-side cross-site scripting and SQL injection vulnerabilities. We present an evaluation of the framework, and demonstrate both the coverage and correctness of our sanitization functions. Finally, experimental results suggest that web applications developed using this framework perform competitively with applications developed using traditional frameworks.

Keywords: Web applications, strongly typed languages, functional languages, cross-site scripting, SQL injection.

1 Introduction

In the last decade, web applications have become an extremely popular means of providing services to large numbers of users. Web applications are relatively easy to develop, the potential audience of a web application is a significant proportion of the planet's population [36, 10], and development frameworks have evolved to the point that web applications are approaching traditional thick-client applications in functionality and usability.

Unfortunately, web applications have also been found to contain many security vulnerabilities [40]. Web applications are also widely accessible and often serve as an interface to large amounts of sensitive data stored in back-end databases. Due to these factors, web applications have attracted much attention from cyber-criminals. Attackers commonly exploit web application vulnerabilities to steal confidential information [41] or to host malware in order to build botnets, both of which can be sold to the highest bidder in the underground economy [46].

By far, the most prevalent security vulnerabilities present in web applications are cross-site scripting (XSS) and SQL injection vulnerabilities [42]. Cross-site scripting vulnerabilities are introduced when an attacker is able to inject malicious scripts into web content to be served to other clients. These scripts then execute with the privileges of the web application delivering the content, and can be used to steal authentication credentials or to install malware, among other nefarious objectives. SQL injections occur when malicious input to a web application is allowed to modify the structure of queries issued to a back-end database. If successful, an attacker can typically bypass authentication procedures, elevate privileges, or steal confidential information.

Accordingly, much research has focused on detecting and preventing security vulnerabilities in web applications. One approach is to deploy web application firewalls (WAFs), usually incorporating some combination of misuse and anomaly detection techniques, in order to protect web applications from attack [6, 14, 8, 29, 45]. Anomaly detection approaches are attractive due to their black-box approach; they typically require no *a priori* knowledge of the structure or implementation of a web application in order to provide effective detection.

Another significant focus of research has been on applying various static and dynamic analyses to the source code of web applications in order to identify and mitigate security vulnerabilities [21, 33, 25, 2, 7, 50]. These approaches have the advantage that developers can con-

tinue to create web applications using traditional languages and frameworks, and periodically apply a vulnerability analysis tool to provide a level of assurance that no security-relevant flaws are present. Analyzing web applications is a complex task, however, as is the interpretation of the results of such security tools. Additionally, several approaches require developers to specify security policies to be enforced in a specialized language.

A more recent line of research has focused on providing client-side protection by enforcing security policies within the web browser [43, 22, 13]. These approaches show promise in detecting and preventing client-side attacks against newer web applications that aggregate content from multiple third parties, but the specification of policies to enforce is generally left to the developer.

In this paper, we propose a different approach to web application security. We observe that cross-site scripting and SQL injection vulnerabilities can be viewed as a failure on the part of the web application to enforce a separation of the *structure* and the *content* of documents and database queries, respectively, and that this is a result of treating documents and queries as untyped sequences of bytes. Therefore, instead of protecting or analyzing existing web applications, we describe a framework that strongly types both documents and database queries. The framework is then responsible for automatically enforcing a separation between structure and content, as opposed to the *ad hoc* sanitization checks that developers currently must implement. Consequently, the integrity of documents and queries generated by web applications developed using our framework are automatically protected, and thus, *by construction*, such web applications are not vulnerable to server-side cross-site scripting and SQL injection attacks.

To illustrate the problem at hand, consider that HTML or XHTML documents to be presented to a client are typically constructed by concatenating strings. Without additional type information, a web application framework has no means of determining that the following operations could lead to the introduction of a cross-site scripting vulnerability:

```
String result = "<div>" + userInput + "</div>";
```

The key intuition behind our work is that because both documents and database queries are strongly typed in our framework, the framework can distinguish between the structure (`<div>` and `</div>`) and the content (`userInput`) of these critical objects, and enforce their integrity automatically.

In this work, we leverage the advanced type system of Haskell, since it offers a natural means of expressing the typing rules we wish to impose. In principle, however, a similar framework could be implemented in any

language with a strong type system that allows for some form of multiple inheritance (e.g., Java or C#).

In summary, the main contributions of this paper are the following:

- We identify the lack of typing of web documents and database queries as the underlying cause of cross-site scripting and SQL injection vulnerabilities.
- We present the design of a web application development framework that automatically prevents the introduction of cross-site scripting and SQL injection vulnerabilities by strongly typing both web documents and database queries.
- We evaluate our prototype web application framework, demonstrate the coverage and correctness of its sanitization functions, and show that applications under our framework perform competitively with those using existing frameworks.

The remainder of this paper is structured as follows. Section 2 presents the design of a strongly typed web application framework. The specification of documents under the framework and how their integrity is enforced is discussed in Section 3, and similarly for SQL queries in Section 4. Section 5 evaluates the design of the framework, and demonstrates that web applications developed under this framework are free from certain classes of vulnerabilities. Related work is discussed in Section 6. Finally, Section 7 concludes and presents avenues for further research.

2 Framework design

At a high level, the web application framework is composed of several familiar components. A web server component processes HTTP requests from web clients and forwards these requests in an intermediate form to the application server based on one of several configuration parameters (e.g., URL path prefix). These requests are directed to one of the web applications hosted by the application server. The web application examines any parameters to the request, performs some processing during which queries to a back-end database may be executed, and generates a document. Note that in the following, the terms “document” or “web document” shall generically refer to any text formatted according to the HTML or XHTML standards. This document is then returned down the component stack to the web server, which sends the document as part of an HTTP response to the web client that originated the request. A graphical depiction of this architecture is given in Figure 1.

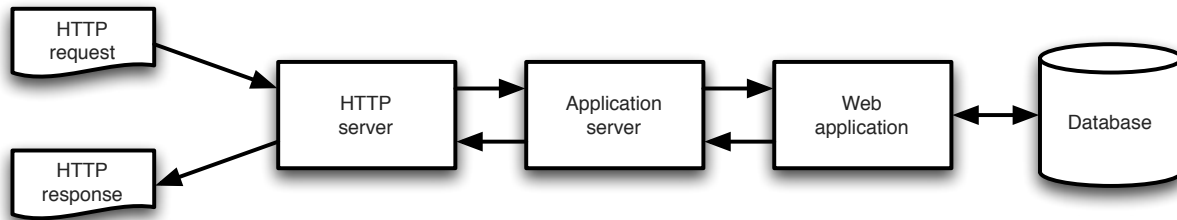


Figure 1: Architectural overview of the web application framework.

Web applications developed for our framework are structured as a set of functions with access to a combination of configuration data and application state. More precisely, web applications execute inside the App monad. Monads are a category theoretic construction that have found wide application in Haskell to sequence actions or isolate code that can produce side effects.¹ For the purposes of our framework, we use the App monad to thread implicit state through the functions comprising a web application, and to provide a controlled interface to potentially dangerous functions. In particular, the App monad itself is structured as a stack of monad transformers that provide a functional interface to a read-only configuration type `AppConfig`, a read-write application state type `AppState`, and filtered access to the IO monad. The definitions for `AppConfig` and `AppState` are given in Figures 2 and 3.

```

data AppConfig = AppConfig {
  appCfgPort :: Int,
  appCfgPrefix :: String,
  appCfgRoutes :: RouteMap,
  appCfgFileRoot :: FilePath,
  appCfgDBConn :: Connection,
  appCfgDBStmts :: StmtMap
}

```

Figure 2: Definition for the `AppConfig` type.

The `AppConfig` type holds static information relating to the configuration of the application, including the port on which to listen for HTTP requests and the root directory of static files to serve from the filesystem. Of particular interest, however, are the `RouteMap` and `StmtMap` fields. The `RouteMap` type describes how URL paths are mapped to values of type `DocumentGen`, which are simply functions that generate documents within the App monad. In addition, the `RouteMap` type contains a default `DocumentGen` type that specifies an error page. Given an incoming HTTP request destined for a partic-

¹For further information on monads, please refer to [37, 48].

ular web application, the application server uses that application’s `RouteMap` type to determine the proper function to call in order to generate the document to be returned to the client.² Finally, the `StmtMap` type associates unique database query identifiers to prepared statements that can be executed by a document generator.

```

data AppState = AppState {
  appStClient :: Maybe SockAddr,
  appStUrl :: Maybe Url
}

```

Figure 3: Definition for the `AppState` type.

The `AppState` type contains mutable state that is specific to each request for a document. In particular, one field records information indicating the source of the request. Additionally, another field records the URL that was requested, including any parameters that were specified by the client. More complex state types that hold additional information (e.g., cached database queries or documents) are possible, however.

3 Document structure

In this section, we introduce the means by which documents are specified under the framework. Then, we discuss how these specifications allow the framework to automatically contain the potentially harmful effects of dynamic data.

3.1 Document specification

Once an appropriate route from the `RouteMap` structure has been selected by the application server, the associated document generator function is executed within the context of the App monad (i.e., with access to the

²This construction is similar to the “routes” packages present in popular web development frameworks such as Rails [18] and Pylons [4].

```

data Document = Document {
  docType :: DocumentType,
  docHead :: DocumentHead,
  docBody :: DocumentBody
}

data DocumentType = DOC_TYPE_HTML_4_01_STRICT
  | DOC_TYPE_HTML_4_01_TRANS
  | ...
  | DOC_TYPE_XHTML_1_1

data DocumentHead = DocumentHead {
  docTitle :: String,
  docLinks :: [Node],
  docScripts :: [Node],
  docBaseUrl :: Maybe Url,
  docBaseTarget :: Maybe Target,
  docProfile :: [Url]
}

data DocumentBody = DocumentBody {
  docBodyNode :: Node
}

```

Figure 4: Definition for the Document type.

configuration and current state of the application). The document generator function processes the request from the application server and returns a variable of type `Document`. The definition of the `Document` type and its constituent types are shown in Figure 4.

As is evident, documents in our framework are not represented as an unstructured stream of bytes. Rather, the structure of the `Document` type closely mirrors that of parsed HTML or XHTML documents. The `DocumentType` field indicates the document's type, such as "HTML 4.01 Transitional" or "XHTML 1.1". The `DocumentHead` type contains information such as the title and client-side code to execute. The `DocumentBody` type contains a single field that represents the root of a tree of nodes that represent the body of the document.

Each node in this tree is an instantiation of the `Node` type. Each `Node` instantiation maps to a distinct (X)HTML element, and records the set of possible properties of that element. For instance, the `TextNode` data constructor creates a `Node` that holds a text string to be displayed as part of a document. The `AnchorNode` data constructor, on the other hand, creates a `Node` that holds information such as the `href` attribute, `rel` attribute, and a list of child nodes corresponding to the text or other elements that comprise the "body" of the link. A partial definition of the `Node` type is presented in Figure 5.

With this construction, the entire document produced by a web application in our framework is strongly typed. Instead of generating a document as a byte stream, doc-

```

data Node = TextNode {
  nodeText :: String
}
  | AnchorNode {
    anchorAttrs :: NodeAttrs,
    anchorHref :: Maybe Url,
    anchorRel :: Maybe Relationship,
    anchorRev :: Maybe Relationship,
    anchorTarget :: Maybe Target,
    anchorType :: Maybe MimeType,
    anchorCharset :: Maybe CharSet,
    anchorLang :: Maybe Language,
    anchorName :: Maybe AttrValue,
    anchorShape :: Maybe Shape,
    anchorCoords :: Maybe Coordinates,
    anchorNodes :: [Node]
  }
  | DivNode {
    divAttrs :: NodeAttrs,
    divNodes :: [Node]
  } ...

```

Figure 5: Sample Node definitions.

ument structure is explicitly encoded as a tree of nodes. Furthermore, each element and element attribute has an associated type that constrains, to one degree or another, the range of possible values that can be represented. For instance, the `MimeType`, `CharSet`, and `Language` types are examples of enumerations that strictly limit the set of possible values the attribute can take to legal values. Standard (X)HTML element attributes (e.g., `id`, `class`, `style`) are represented with the `NodeAttr` type. Optional attributes are represented using either the `Maybe` type,³ or as an empty list if multiple elements are allowed.

Note that it is possible for a `Document` to represent an (X)HTML document that is not necessarily consistent with the respective W3C grammars that specify the set of well-formed documents. One example is that any `Node` instantiation may appear as the child of any other `Node` that can hold children, which violates the official grammars in several instances. Strict conformance with the W3C standards is not, however, our goal.⁴

Instead, the typing scheme presented here allows our framework to specify a separation between the *structure* and the *content* of the documents a web application generates. More precisely, the dynamic data that enters a web application as part of an HTTP request (e.g., as a GET or POST parameter) can indirectly influence the structure of a document. For instance, a search request to a web application may result in a variable number of

³`Maybe` allows for the absence of a value, as Haskell does not possess nullable types. For example, the type `Maybe a` can be either `Just "..."` or `Nothing`.

⁴Indeed, standards-conforming documents have been shown to be difficult to represent in a functional language [12].

```

class Render a where
    render :: a -> String

instance Render AttrValue where
    render = quoteAttr

quoteAttr :: AttrValue -> String
quoteAttr a = foldl' step [] (attrValue a)

step acc c | c == '<' = acc ++ "&lt;"
           | c == '>' = acc ++ "&gt;"
           | c == '&' = acc ++ "&amp;"
           | c == '"' = acc ++ "&quot;"
           | otherwise = acc ++ [c]

```

Figure 6: Render typeclass definition and (simplified) instance example. Here, `quoteAttr` performs a left fold over attribute values using `foldl'`, which applies the `step` function to each character of the string and accumulates the result. The definition of `step` specifies a number of *guards*, where `| c == '<'` is a condition that must be satisfied for the statement `acc ++ "<";` to execute. This statement simply appends the string `"<";` to `acc`, the accumulator, in order to build a new, sanitized string. If no guard condition is satisfied, the character is appended without conversion.

table rows in the generated document depending on the number of results returned from a database query. Due to our framework, however, client-supplied data cannot *directly* modify the structure of the document in such a way that a code injection can occur.

3.2 Enforcing document integrity

Once a `Document` has been constructed by the web application in response to a client request, it is returned to the application server. The application server is responsible for converting this data structure into a format the client can understand – that is, it must *render* the document into a stream of bytes representing an (X)HTML document. Consequently, the set of types that can comprise a `Document` are instances of the `Render` typeclass, shown in Figure 6.⁵

The `Render` type class specifies that any instance of the class must implement the `render` function. From the type signature, the semantics of the function are clear: `render` converts an instance type into a string representation suitable for presentation to a client. Crucially, for our purposes, the `Render` type class is also responsible for enforcing the integrity of a document's structure.

⁵Haskell typeclasses are roughly similar to Java interfaces, in that they specify a function interface that all instances (in Java, implementors) must provide.

As an example, Figure 6 presents a simplified `render` definition for the `AttrValue` type that is used to indicate element attribute strings that may assume (almost) arbitrary values. In order to preserve the integrity of the document, an attribute value must not contain certain characters that would allow an attacker to inject malicious code into the document. Consider, for instance, the following element:

```
<input type="hidden" name="h1" value="..." />
```

Now, suppose an attacker submitted the following string as part of a request such that it was reflected to another client as the value of the hidden input field:

```

"/>
<script src="http://example.com/malware.js">
</script>
<span id="

```

The result would be the following:

```

<input type="hidden" name="h1" value=""/>
<script src="http://example.com/malware.js">
</script>
<span id=""/>

```

To prevent such an injection from occurring, the `render` function for the `AttrValue` class applies a sanitization function on the string wrapped by `AttrValue`. Any occurrence of an unsafe character is replaced by an equivalent HTML entity encoding that can safely appear as part of an attribute value.⁶ Similar `render` functions are defined for the set of types that can comprise a `Document`.

Therefore, to prepare a `Document` as part of an HTTP response to a client, the application server applies the `render` function to the document, which recursively converts the data structure into an (X)HTML document. As part of this process, the content of the document is sanitized by type-specific `render` functions, ensuring that client-supplied input to the web application cannot modify the document structure in such a way as to result in a client-side code injection.

4 SQL query structure

Similarly to the case of documents, SQL queries are given structure in our framework through the application of strong typing rules that control how the structure of the query can be combined with dynamic data. In this section, we examine the structure of SQL queries and discuss two mechanisms by which SQL query integrity is enforced under the framework.

⁶In the real implementation, the sanitization function is somewhat more complex, as there are multiple encodings by which an unsafe character can be injected. The example function given here is simplified for the purposes of presentation.

```

INSERT INTO users(login, passwd)
VALUES(?, ?)

SELECT * FROM users
WHERE login='admin' AND passwd='test'

UPDATE users SET passwd='$passwd'
WHERE login='$login'

```

Figure 7: Examples of SQL queries.

4.1 Query specification

SQL queries, as shown in Figure 7, are composed of clauses, predicates, and expressions. For instance, a clause might be `SELECT *` or `UPDATE users`. An example of a predicate is `login='admin'`, where `'admin'` is an expression. Clauses, predicates, and expressions are themselves composed of static tokens, such as keywords (`SELECT`) and operators (`=`), and dynamic tokens, such as table identifiers (`users`) or data values (`'admin'`).

Typically, the structure of a SQL query is fixed.⁷ Specifically, a query will have a static keyword denoting the operation to perform, will reference a static set of tables and fields, and specify a fixed set of predicates. Generally, the only components of a query that change from one execution to the next are data values, and, even then, their number and placement remain fixed.

SQL injection attacks rely upon the ability of the attacker to modify the structure of a query in order to perform a malicious action. When SQL queries are constructed using string operations without sufficient sanitization applied to user input, such attacks become trivial. For instance, consider the `UPDATE` query shown in Figure 7. If an attacker were to supply the value `"quux' OR login='admin'"` for the `$login` variable, the following query would result:

```

UPDATE users SET passwd='foo'
WHERE login='quux' OR login='admin'

```

Because the attacker was able to inject single quotes, which serve as delimiters for data values, the structure of the query was changed, resulting in a privilege escalation attack.

4.2 Integrity enforcement with static query structure

In contrast to the case of document integrity enforcement, a well-known solution exists for specifying SQL

⁷This is not always the case, but the case of dynamic query structure will be considered later in this section.

```

SELECT * FROM users WHERE login=? AND passwd=?
UPDATE users SET passwd=? WHERE login=?

```

Figure 8: Examples of prepared statements, where “?” characters serve as placeholders for data substitution.

query structure: prepared statements. Prepared statements are a form of database query consisting of a parameterized query template containing placeholders where dynamic data should be substituted. An example is shown in Figure 8, where the placeholders are signified by the “?” character.

A prepared statement is typically parsed and constructed prior to execution, and stored until needed. When an actual query is to be issued, variables that may contain client-supplied data are *bound* to the statement. Since the query has already been parsed and the placeholders specified, the structure of the query cannot be modified by the traditional means of providing malicious input designed to be interpreted as part of the query. In the case of the injection attack described previously, the result would be as the following (note that the injected single quotes have been escaped):

```

UPDATE users SET passwd='foo'
WHERE login='quux\'' OR login='admin'

```

From our perspective, the query has been typed as a composition of static and dynamic elements; it is exactly this distinction between structure and content that we wish to enforce. Haskell’s database library (HDBC), as do most other languages, supports the use of prepared statements. Therefore, the framework exports functions that allow a web application to associate prepared statements with a unique identifier in the `AppConfig` type. During request processing, a document generator can then retrieve a prepared statement using the identifier, bind values to it, and execute queries that are not vulnerable to injection attacks.

One detail remains, however. The HDBC library also provides functions allowing traditional *ad hoc* queries that are assembled as concatenations of strings to be executed. Without any other modification to the framework, a web application developer would be free to directly call these functions and bypass the protections afforded by the framework. Therefore, an additional component is required to encapsulate the HDBC interface and prevent execution of these unsafe functions. This component takes the form of a monad transformer `AppIO`, which simply wraps the `IO` monad and exposes only those functions that are considered safe to execute. The structure of this stack is shown in Figure 9. In this environment, within which all web applications using the framework operate, unsafe database execution functions are inaccessible, since they will fail to type-check. Thus, assuming

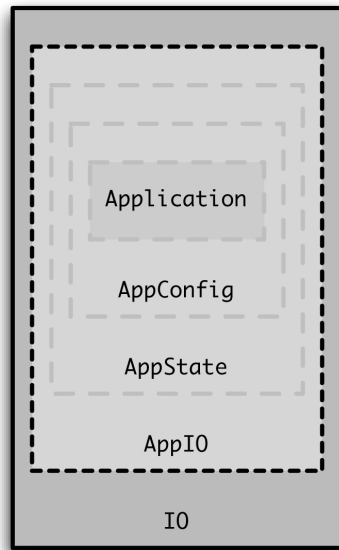


Figure 9: Graphical representation of the monad stack within which framework applications execute. The AppIO monad encapsulates applications, preventing them from calling unsafe functions within the IO monad.

the correctness of the HDBC prepared statement interface, web applications developed using the framework are not vulnerable to SQL injection.

4.3 Integrity enforcement with dynamic query structure

Though most SQL queries possess a fixed structure, there does exist a small class of SQL queries that exhibit dynamic structure. For instance, many SQL database implementations provide a set membership operator, where queries of the form

```
SELECT * FROM users WHERE
login IN ('admin', 'developer', 'tester')
```

can be expressed. In this case, the size of the set of data values can often change at runtime. Another example is the case where the structure of queries is determined by the user, for instance through a custom search form where many different combinations of predicates can be dynamically expressed. Unfortunately, since these queries cannot be represented using prepared statements, they cannot be protected using the monadic encapsulation technique described previously.

Therefore, a second database interface is exposed by the framework to the application developers. Instead of relying upon prepared statements, this interface allows developers to dynamically construct queries as a tree of algebraic data types as in the case of web documents.

```
data Select = Select {
  sFields :: [Expr],
  sTables :: [Expr],
  sCons :: Maybe Expr,
  sGrpFields :: [Expr],
  sGrpCons :: Maybe Expr,
  sOrdFields :: [Expr],
  sLimit :: Maybe Int,
  sOffset :: Maybe Int,
  sDistinct :: Bool
}

data Expr = EXPR_TABLE Table
          | EXPR_FIELD Field
          | EXPR_DATA String
          | EXPR_NOT Expr
          | EXPR_OR Expr Expr
          | EXPR_AND Expr Expr
          | ...

data Table = Table {
  tName :: String,
  tAlias :: Maybe String
}

data Field = Field {
  fName :: String,
  fAlias :: Maybe String
}
```

Figure 10: Definition for the Select type.

Figure 10 shows an example of the type representing a SELECT query.

To populate instances of these types, the interface provides a set of combinators, or higher-order functions, that can be chained together. These combinators, which assume names similar to SQL keywords, implement an embedded domain-specific language (DSL) that allows application developers to naturally specify dynamic queries within the framework. For instance, a query could be constructed using the following sequence of function invocations:

```
qSelect [qField "*"] >>=
qFrom [qTable "users"] >>=
qWhere (((qField "login") == (qData "admin")) &&
((qField "passwd") == (qData "test")))
```

Similar to the case of the Document type, queries constructed in this manner are transformed into raw SQL statements solely by the framework.⁸ Therefore, the types that represent queries also implement the Render

⁸Note that, as in the case of web documents, we do not attempt to enforce the generation of correct SQL, but rather focus on preventing attacks by preserving query structures specified by the developer.

Context	Semantics
Document nodes	Conversion to static string
Document node attributes	Encoding of HTML entities
Document text	Encoding of HTML entities
URL components	Encoding of HTML entities, percent encoding
SQL static value	Removal of spaces, comments, quotes
SQL data value	Escaping of quotes

Table 1: Example contexts for which specific sanitization functions are applied, and the semantics of those sanitization functions under various encodings.

typeclass. Consequently, sanitization functions must be applied to each of the fields comprising the query types, such that the intended structure of the query cannot be modified. This can be accomplished by enforcing the conditions that no data value may contain an unescaped single quote, and that all remaining query components may not contain spaces, single quotes, or character signifying the beginning of a comment. Assuming that these sanitization functions are correct, this construction renders applications developed under the framework invulnerable to SQL injection attacks while allowing for more powerful query specifications.

5 Evaluation

To demonstrate that web applications developed using our framework are secure by construction from server-side XSS and SQL injection vulnerabilities, we conducted an evaluation of the system. First, we demonstrate that all dynamic content contained in a `Document` must be sanitized by an application of the `render` function, and that a similar condition holds for dynamically-generated SQL queries. Then, we provide evidence that the sanitization functions themselves are correct – that is, they successfully strip or encode unsafe characters. We also verify that the prepared statement library prevents injections, as expected. Finally, to demonstrate the viability of the framework, an experiment to evaluate the performance of a web application developed using the framework is conducted.

5.1 Sanitization function coverage

The goal of the first experiment was to justify the claim that all dynamic content contained in a `Document` or query type must be sanitized prior to presentation to the client that originated the request. To accomplish this, a static control flow analysis of the framework was performed. Figure 11 presents a control flow graph of the application server in a simplified form, where function calls are sequenced from left to right. Of particular inter-

est is the `renderDoc` function, which retrieves the appropriate document generator given a URL path, executes it in the call to `route`, sanitizes it by applying `render`, and creates an HTTP response by calling `make200`. The sanitized document is then returned to `procRequest`, which writes it to the client. Therefore, the entire process of converting the document to a byte stream for presentation to the client is solely due to the recursive `render` application. Similarly, because the only interface exposed to applications to execute SQL queries are `execStmt` and `execPrepStmt` from within the `App` monad, queries issued by applications under the framework must be sanitized either by the framework or the `HDBC` prepared statement functions.

Figure 12 displays a subset of the full control flow graph depicting an instance of the `render` function for the `AnchorNode` `Node` instantiation. For clarity of presentation, multiple calls to `render` and `maybeRenderAttr` have been collapsed into single nodes. Recall from Figure 5 that the definition of `AnchorNode` does not contain any bare strings; instead, each field of the type is either itself a composite type, or an enumeration for which a custom `render` function is defined. Since no other string conversion function is applied in this subgraph, we conclude that all data contained in an `AnchorNode` variable must be filtered through a sanitization function.

The analysis of this single case generalizes to the set of all types that can comprise a `Document` or query type. In total, 163 distinct sanitization function definitions were checked to sanitize the contexts shown in Table 1. For each function, our analysis found that no irreducible type was concatenated to the document byte stream without first being sanitized.

5.2 Sanitization function correctness

The goal of this experiment is to determine whether the sanitization functions employed by the framework are correct (i.e., whether all known sequences of dangerous characters are stripped or encoded). To establish this, we applied a dynamic test-driven approach using the

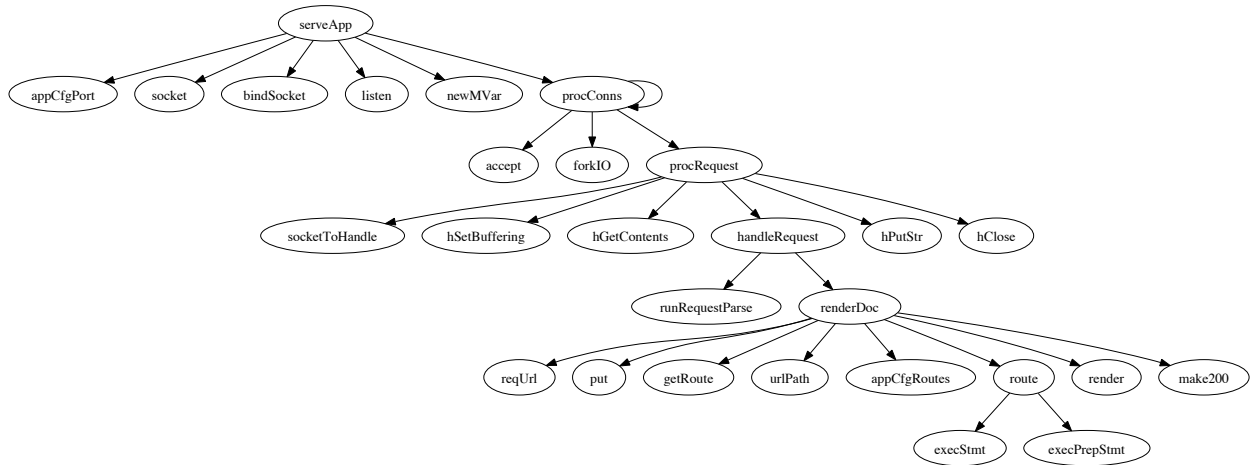


Figure 11: Simplified control flow graph for application server.

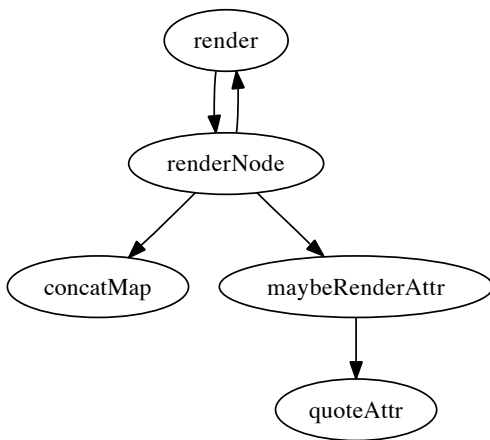


Figure 12: Example control flow graph for Render Node instance.

QuickCheck property testing library [9]. QuickCheck allows a developer to specify invariants in an embedded language that should hold for a given set of functions. The library then automatically generates a set of random test cases, and checks that the invariants hold for each test. In our case, we selected invariants based upon known examples of XSS [44] and SQL injection attacks [15]. In addition, we introduced modifications of the invariants that account for different popular document encodings, since these encodings directly affect how browser parsers interpret the sequences of bytes that comprise a document.

Since the coverage of the sanitization functions has been established by the control flow analysis, we focused our invariant testing on the low-level functions responsible for processing string data. In particular, we specified invariants for 7 functions that are responsible for sani-

tizing (X)HTML content, element attributes, and various URL components.⁹ An example invariant specification is shown in Figure 13.

```
propAttrValueSafe :: AttrValue -> Bool
propAttrValueSafe input =
  (not $ elem '<' output) &&
  (not $ elem '>' output) &&
  (not $ elem '&' $ stripEntities output) &&
  (not $ elem '"' output) where
    output = render input
```

Figure 13: Simplified example sanitization function invariant specification. Here, `propAttrValueSafe` is a conjunction of predicates, where `not $ elem c output` specifies that the character `c` should not be an element of the output of `render` in this context. Since “&” is used to indicate the beginning of an HTML entity (e.g., `&`), the `stripEntities` function ensures that ampersands may only appear in this form.

For each of the sanitization functions, we first tested the correctness of the invariants by checking that they were violated over a set of 100 strings corresponding to real-world cross-site scripting, command injection, and other code injection attacks. Then, for each sanitization function, we generated 1,000,000 test cases of random strings using the QuickCheck library. In all cases, the invariants were satisfied.

In addition to performing invariant testing on the set of document sanitization functions, we also applied a similar testing process to the sanitization of query types described in Section 4.3. Finally, we applied manual invari-

⁹The 163 functions noted above eventually apply one of these 7 context-specific sanitization functions for web documents.

ant testing on the JDBC prepared statement interface. In all cases, the invariants on the integrity of the queries and the database itself held.

5.3 Framework performance

In this experiment, we compared the performance of a web application developed using our framework to similar applications implemented using other frameworks. In particular, we developed a small e-commerce site with a product display page, cart display page, and checkout page under our framework, using the Pylons framework 0.9.7 [4], and as a Java servlet using Tomcat 6.0.18.¹⁰ Each application was backed by a SQLite database containing product information. The application servers were hosted on a server running Ubuntu Server 8.10 with dual Intel Core 2 Duo CPUs, 2 GB of RAM, and 1000BaseT Ethernet network interfaces. The `httperf` [20] web server benchmarking tool was deployed on a similar server to generate load for each application.

Figure 14 presents averaged latency and throughput plots for 8 benchmarking runs for each framework tested. In each run, the number of concurrent clients issuing requests was varied, and the average response latency in milliseconds and the aggregate throughput in kilobytes was recorded. In this experiment, our framework performed competitively compared to Pylons and Tomcat, performing somewhat better than Pylons in both latency and throughput scaling, and vice versa for Tomcat. In particular, the latency plot shows that our framework scales significantly better with the number of clients than the Pylons framework. Unfortunately, our framework exhibited approximately a factor of two increase in latency compared to the Tomcat application. Cost-center profiling revealed that this is mainly due to the overhead of list-based `String` operations in Haskell,¹¹ though this could be ameliorated by rewriting the framework to prefer the lower-overhead `ByteString` type. Therefore, it is not unreasonable to assume that web applications developed using our framework would exhibit acceptable performance behavior in the real world.

5.4 Discussion

The security properties enforced by this framework are effective at guaranteeing that applications are not vulnerable to server-side XSS and SQL injection. There are limitations to this protection that need to be highlighted, however, and we discuss these here.

¹⁰Pylons is a Python-based framework that is similar in design to Ruby on Rails, and is used to implement a variety of well-known web applications (e.g., Reddit (<http://reddit.com/>)).

¹¹Strings are represented as lists of characters in Haskell – that is, `type String = [Char]`.

First, web applications can, in some cases, be vulnerable to *client-side* XSS injections, or DOM-based XSS, where the web application can potentially not receive any portion of such an attack [28]. This can occur when a client-side script dynamically updates the DOM after the document has been rendered by the browser with data controlled by an attacker. In general, XSS attacks stemming from the misbehavior of client-side code within the browser are not addressed by the framework in its current form.

Recently, a new type of XSS attack against the content-sniffing algorithms employed by web browsers has been demonstrated [5]. In this attack, malicious non-HTML files that nevertheless contain HTML fragments and client-side code are uploaded to a vulnerable web application. When such a file is downloaded by a victim, the content-sniffing algorithm employed by the victim's browser can potentially interpret the file as HTML, executing the client-side code contained therein, resulting in an XSS attack. Consequently, our framework implements the set of file upload filters recommended by the authors of [5] to prevent content-sniffing XSS. Since, however, the documents are supplied by users and not generated by the framework itself, the framework cannot guarantee that it is immune to such attacks.

Finally, CSS stylesheets and JSON documents can also serve as vectors for XSS attacks. In principle, these documents could be specified within the framework using the same techniques applied to (X)HTML documents, along with context-specific sanitization functions. In the case of CSS stylesheets that are uploaded to a web application by users, additional sanitization functions could be applied to strip client-side code fragments. However, the framework in its current form does not address these vectors.

6 Related work

An extensive literature exists on the detection of web application vulnerabilities. One of the first tools to analyze server-side code for vulnerabilities was WebSSARI [21], which performs a taint propagation analysis of PHP in order to identify potential vulnerabilities, for which runtime guards are inserted. Nguyen-Tuong *et al.* proposed a precise taint-based approach to automatically hardening PHP scripts against security vulnerabilities in [39]. Livshits and Lam [33] applied a points-to static analysis to Java-based web applications to identify a number of security vulnerabilities in both open-source programs and the Java library itself. Jovanovic *et al.* presented Pixy, a tool that performs flow-sensitive, interprocedural, and context-sensitive data flow analysis to detect security vulnerabilities in PHP-based web applications [25]; Pixy was later enhanced with precise alias analysis to improve

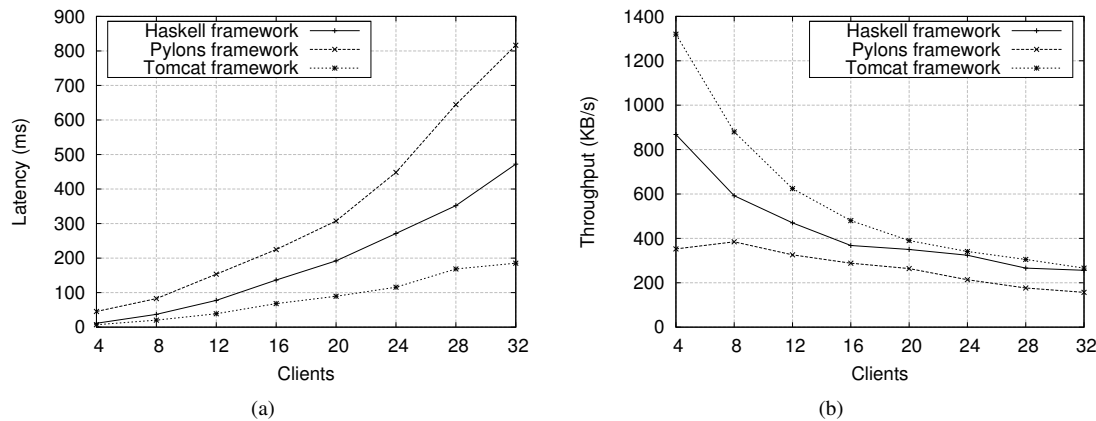


Figure 14: Latency and throughput performance for the Haskell, Pylons, and Tomcat-based web applications.

the accuracy of the technique [26]. A precise, sound, and fully automated technique for detecting modifications to the structure of SQL queries was described by Wassermann and Su in [49]. Balzarotti *et al.* observed that more complex vulnerabilities in web applications can manifest themselves as interactions between distinct modules comprising the application, and proposed MiMoSA to perform multi-module vulnerability analysis of PHP applications [2]. In [7], Chong *et al.* presented SIF, a framework for developing Java servlets that enforce legal information flows specified by a policy language. A syntactic technique of string masking is proposed by Johns *et al.* in [23] in order to prevent code injection attacks in web applications. Lam *et al.* described another information flow enforcement system using PQL, and additionally propose the use of a model checker to generate test cases for identified vulnerabilities [30]. In [1], Balzarotti *et al.* applied a combination of static and dynamic analysis to check the correctness of web application sanitization functions. Wassermann and Su applied a combination of taint-based information flow and string analysis to enforce effective sanitization policies against cross-site scripting in [50]. Nadji *et al.* propose a similar notion of document structure integrity in [38], using a combination of web application code randomization and runtime tracking of untrusted data on both the server and the browser. Finally, Google's ctemplate [17], a templating language for C++, and Django [11], a Python-based web application framework, include an Auto-Escape feature that allows for context-specific sanitization of web documents, while Microsoft's LINQ [35] is an approach for performing language-integrated data set queries in the .NET framework.

The approach described in this work differs from the above server-side techniques in several respects. First, an advantage of several of the above techniques is that they provide greater generality in their enforcement of secu-

rity policies; in particular, SIF allows for the enforcement of complex information flows and uses some of the techniques presented in this work. Our framework, on the other hand, requires neither information flow policy specifications or additional static or dynamic analyses to protect against cross-site scripting or SQL injection vulnerabilities. String masking embodies a similar notion of a separation of code and data for web applications, but is implemented as a preprocessor for existing web applications and allows the possibility for both false positives and false negatives. Django and ctemplate are similar in spirit to this work in that they apply a similar context-sensitive sanitization of documents generated from template specifications. In both cases, however, this sanitization is optional and relies upon a separate document parser, whereas documents in our framework are specified in the language itself. ctemplate in particular has an advantage in that it supports limited sanitization of CSS and JSON documents, though this analysis is not currently based upon a robust parser. Finally, LINQ provides a language-based mechanism for dynamically constructing parameterized queries on arbitrary data sets, including SQL databases, and is therefore similar to the system proposed in this framework. Use of this interface is, however, optional and can be bypassed.

In addition to server-side vulnerability analyses, much work has focused on client-side protection against malicious code injection. The first system to implement client-side protection was due to Kirda *et al.* In [27], the authors presented Noxes, a client-side proxy that uses manual and automatically-generated rules to prevent cross-site scripting attacks. Vogt *et al.* proposed a combination of dynamic data tainting and static analysis to prevent cross-site scripting attacks from successfully executing within a web browser [47]. BrowserShield, due to Reis *et al.*, is a system to download signatures for known cross-site scripting exploits; JavaScript wrappers

that implement signature detection for these attacks are then installed into the browser [43]. Livshits and Erlingsson described an approach to cross-site scripting and RSS attacks by modifying JavaScript frameworks such as Dojo, Prototype, and AJAX.NET in [32]. BEEP, presented by Jim *et al.* in [22], implements a coarse-grained approach to client-side policy enforcement by specifying both black- and white-lists of scripts. Erlingsson *et al.* proposed Mutation-Event Transforms, a technique for enforcing finer-grained client-side security policies by intercepting JavaScript calls that would result in potentially malicious modifications to the DOM [13].

In contrast to the client-side approaches discussed here, our framework does not require a separate analysis to determine whether cross-site scripting vulnerabilities exist in a web application. In the case of web applications that include client-side scripts from untrusted third parties (e.g., mashups), a client-side system such as BEEP or Mutation-Event Transforms can be considered a complementary layer of protection to that provided by our framework.

Several works have studied how the safety of functional languages can be improved. Xu proposed the use of pre/post-annotations to implement extended static checking for Haskell in [51]; this work has been extended in the form of contracts in [52]. Li and Zdancewic demonstrated how general information flow policies could be integrated as an embedded security sublanguage in Haskell in [31]. A technique for performing data flow analysis of lazy higher-order functional programs using regular sets of trees to approximate program state is proposed by Jones and Andersen in [24]. Madhavapeddy *et al.* presented a domain-specific language for securely specifying various Internet packet protocols in [34]. In [16], Finifter *et al.* describe Joe-E, a capability-based subset of Java that allows programmers to write pure Java functions that, due to their referential transparency, admit strong analyses of desirable security properties.

The work presented in this paper differs from those above in that our framework is designed to mitigate specific vulnerabilities that are widely prevalent on the Internet. The generality of our approach could be enhanced, however, by integrating general information flow policies, at the cost of an additional burden on developers.

Several application servers for Haskell have already been developed, most notably HAppS [19]. To the best of our knowledge, however, none of these frameworks implement specific protection against cross-site scripting or SQL injection vulnerabilities. Finally, Elsmann and Larsen studied how XHTML documents can be typed in ML [12]. Their focus, however, is on generating standards-conforming documents; they do not directly address security concerns.

7 Conclusions

In this paper, we have presented a framework for developing web applications that, by construction, are invulnerable to server-side cross-site scripting and SQL injection attacks. The framework accomplishes this by strongly typing both documents and database queries that are generated by a web application, thereby automatically enforcing a separation between structure and content that preserves the integrity of these objects.

We conducted an evaluation of the framework, and demonstrated that all dynamic data that is contained in a document generated by a web application must be subjected to sanitization. Similarly, we show that all SQL queries must be executed in a safe manner. We also demonstrate the correctness of the sanitization functions themselves. Finally, we give performance numbers for representative web applications developed using this framework that compare favorably to those developed in other popular environments.

In future work, we plan to investigate how the framework can be modified to allow developers to specify “safe” transformations of document structure to occur in a controlled manner. Also, we plan to investigate static techniques for verifying the correctness of the sanitization functions in terms of their agreement with invariants extracted from web browser document parsers and database query parsers, for instance using a combination of static and dynamic analyses [3, 5]. Finally, future work will consider how language-based techniques for ensuring document integrity could be applied on the client.

Acknowledgments

The authors wish to thank the anonymous reviewers for their insightful comments. We would also like to thank Adam Barth for providing feedback on an earlier version of this paper. This work has been supported by the National Science Foundation, under grants CCR-0238492, CCR-0524853, and CCR-0716095.

References

- [1] D. Balzarotti, M. Cova, V. V. Felmetzger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy (S&P 2008)*, Oakland, CA, USA, May 2008.
- [2] D. Balzarotti, M. Cova, V. V. Felmetzger, and G. Vigna. Multi-module Vulnerability Analysis of Web-

- based Applications. In *Proceedings of the 2007 ACM Conference on Computer and Communications Security (CCS 2007)*, Alexandria, VA, USA, October 2007.
- [3] D. Balzarotti, W. Robertson, C. Kruegel, and G. Vigna. Improving Signature Testing Through Dynamic Data Flow Analysis. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC 2007)*, Miami Beach, FL, USA, December 2007.
 - [4] B. Bangert and J. Gardner. PylonsHQ. <http://pylonshq.com/>, February 2009.
 - [5] A. Barth, J. Caballero, and D. Song. Secure Content Sniffing for Web Browsers, or How to Stop Papers from Reviewing Themselves. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, USA, May 2009. IEEE Computer Society.
 - [6] Breach Security, Inc. Breach WebDefend. <http://www.breach.com/products/webdefend.html>, January 2009.
 - [7] S. Chong, K. Vikram, and A. C. Myers. SIF: Enforcing Confidentiality and Integrity in Web Applications. In *Proceedings of the 2007 USENIX Security Symposium*, Boston, MA, USA, 2007. USENIX Association.
 - [8] Citrix Systems, Inc. Citrix Application Firewall. <http://www.citrix.com/English/PS2/products/product.asp?contentID=25636>, January 2009.
 - [9] K. Claessen and J. Hughes. Testing Monadic Code with QuickCheck. *ACM SIGPLAN Notices*, 37(12):47–59, 2002.
 - [10] M. de Kunder. The Size of the World Wide Web. <http://www.worldwidewebsite.com/>, May 2008.
 - [11] Django Software Foundation. Django Web Application Framework. <http://www.djangoproject.com/>, June 2009.
 - [12] M. Elsmann and K. F. Larsen. Typing XHTML Web Applications in ML. In *Proceedings of the 6th International Symposium on Practical Aspects of Declarative Languages*, pages 224–238. Springer-Verlag, 2004.
 - [13] U. Erlingsson, B. Livshits, and Y. Xie. End-to-end Web Application Security. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems*, San Diego, CA, USA, 2007. USENIX Association.
 - [14] F5 Networks, Inc. BIG-IP Application Security Manager. <http://www.f5.com/products/big-ip/product-modules/application-security-manager.html>, January 2009.
 - [15] Ferruh Mavituna. SQL Injection Cheat Sheet. <http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/>, June 2009.
 - [16] M. Finifter, A. Mettler, N. Sastry, and D. Wagner. Verifiable Functional Purity in Java. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, pages 161–174, Alexandria, VA, USA, October 2008. ACM.
 - [17] Google, Inc. ctemplate. <http://code.google.com/p/google-ctemplate/>, June 2009.
 - [18] D. H. Hansson. Ruby on Rails. <http://rubyonrails.org/>, February 2009.
 - [19] HAppS LLC. HAppS – The Haskell Application Server. <http://happs.org/>, February 2009.
 - [20] Hewlett Packard Development Company, L.P. httpperf. <http://www.hpl.hp.com/research/linux/httpperf/>, February 2009.
 - [21] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D. Lee, and S.-Y. Kuo. Securing Web Application Code by Static Analysis and Runtime Protection. In *Proceedings of the 13th International Conference on the World Wide Web*, pages 40–52, New York, NY, USA, 2004. ACM.
 - [22] T. Jim, N. Swamy, and M. Hicks. Defeating Script Injection Attacks with Browser-Enforced Emded Policies. In *Proceedings of the 16th International Conference on the World Wide Web*, Banff, Alberta, Canada, May 2007. ACM.
 - [23] M. Johns and C. Beyerlein. SMask: Preventing Injection Attacks in Web Applications by Approximating Automatic Data/Code Separation. In *Proceedings of ACM Symposium on Applied Computing*, Seoul, Korea, March 2007. ACM.
 - [24] N. D. Jones and N. Andersen. Flow analysis of lazy higher-order functional programs. *Theoretical Computer Science*, 375(1–3):120–136, 2007.
 - [25] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P 2006)*, pages 258–263, Oakland, CA, USA, May 2006. IEEE Computer Society.

- [26] N. Jovanovic, C. Kruegel, and E. Kirda. Precise Alias Analysis for Static Detection of Web Application Vulnerabilities. In *Proceedings of the 2006 Workshop on Programming Languages and Analysis for Security*, pages 27–36, Ottawa, Ontario, Canada, 2006. ACM.
- [27] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: A Client-side Solution for Mitigating Cross-Site Scripting Attacks. In *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC 2006)*, Dijon, France, April 2006. ACM.
- [28] A. Klein. DOM Based Cross Site Scripting or XSS of the Third Kind. <http://www.webappsec.org/projects/articles/071105.shtml>, July 2005.
- [29] C. Kruegel, W. Robertson, and G. Vigna. A Multi-model Approach to the Detection of Web-based Attacks. *Journal of Computer Networks*, 48(5):717–738, July 2005.
- [30] M. S. Lam, M. Martin, B. Livshits, and J. Whaley. Securing Web Applications with Static and Dynamic Information Flow Tracking. In *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 3–12, San Francisco, CA, USA, 2008. ACM.
- [31] P. Li and S. Zdancewic. Encoding Information Flow in Haskell. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop*. IEEE Computer Society, 2006.
- [32] B. Livshits and U. Erlingsson. Using Web Application Construction Frameworks to Protect Against Code Injection Attacks. In *Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security*, pages 95–104, San Diego, CA, USA, 2007. ACM.
- [33] B. Livshits and M. Lam. Finding Security Errors in Java Programs with Static Analysis. In *Proceedings of the 14th USENIX Security Symposium (USENIX Security 2005)*, pages 271–286. USENIX Association, August 2005.
- [34] A. Madhavapeddy, A. Ho, T. Deegan, D. Scott, and R. Sohan. Melange: Creating a “Functional Internet”. In *Proceedings of the 2nd ACM European Conference on Computer Systems*, pages 101–114, Lisbon, Portugal, April 2007. ACM.
- [35] Microsoft, Inc. LINQ. <http://msdn.microsoft.com/en-us/netframework/aa904594.aspx>, June 2009.
- [36] Miniwatts Marketing Group. World Internet Usage Statistics. <http://www.internetworldstats.com/stats.htm>, May 2008.
- [37] E. Moggi. Notions of Computation and Monads. *Information and Computation*, 93(1):55–92, 1991.
- [38] Y. Nadji, P. Saxena, and D. Song. Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense. In *Proceedings of the Network and Distributed System Security Symposium*, February 2009.
- [39] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shifley, and D. Evans. Automatically Hardening Web Applications Using Precise Tainting. In *Proceedings of the 2005 International Information Security Conference*, pages 372–382, 2005.
- [40] Ofer Shezaf and Jeremiah Grossman and Robert Auger. Web Hacking Incidents Database. <http://www.xiom.com/whid-about>, January 2009.
- [41] Open Security Foundation. DLDOS: Data Loss Database – Open Source. <http://datalossdb.org/>, January 2009.
- [42] Open Web Application Security Project (OWASP). OWASP Top 10 2007. http://www.owasp.org/index.php/Top_10_2007, February 2009.
- [43] C. Reis, J. Dunagan, H. J. Wang, and O. Dubrovsky. BrowserShield: Vulnerability-Driven Filtering of Dynamic HTML. *ACM Transactions on the Web*, 1(3):11, 2007.
- [44] Robert Hansen (RSnake). XSS (Cross Site Scripting) Cheat Sheet. <http://ha.ckers.org/xss.html>, June 2009.
- [45] W. Robertson, G. Vigna, C. Kruegel, and R. A. Kemmerer. Using Generalization and Characterization Techniques in the Anomaly-based Detection of Web Attacks. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2006)*, San Diego, CA, USA, February 2006.
- [46] Symantec, Inc. Symantec Report on the Underground Economy – July 07 – June 08. http://eval.symantec.com/mktginfo/enterprise/white-papers/b-whitepaper-underground-economy-report_11-2008-14525717.en-us.pdf, November 2008.
- [47] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross Site Scripting

Prevention with Dynamic Data Tainting and Static Analysis. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2007)*, February 2007.

- [48] P. Wadler. The Essence of Functional Programming. In *Proceedings of the 19th Annual Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, NM, USA, 1992. ACM.
- [49] G. Wassermann and Z. Su. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. *ACM SIGPLAN Notices*, 42(6):32–41, April 2007.
- [50] G. Wassermann and Z. Su. Static Detection of Cross-Site Scripting Vulnerabilities. In *Proceedings of the 2008 International Conference on Software Engineering (ICSE 2008)*, pages 171–180, Leipzig, Germany, 2008. ACM.
- [51] D. N. Xu. Extended Static Checking for Haskell. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell*, pages 48–59, Portland, OR, USA, 2006. ACM.
- [52] D. N. Xu, S. P. Jones, and K. Claessen. Static Contract Checking for Haskell. In *Proceedings of the 36th Annual ACM Symposium on the Principles of Programming Languages*, pages 41–52, Savannah, GA, USA, 2009. ACM.

Vanish: Increasing Data Privacy with Self-Destructing Data

Roxana Geambasu

Tadayoshi Kohno

Amit A. Levy

Henry M. Levy

University of Washington
{roxana, yoshi, levya, levy}@cs.washington.edu

Abstract

Today's technical and legal landscape presents formidable challenges to personal data privacy. First, our increasing reliance on Web services causes personal data to be cached, copied, and archived by third parties, often without our knowledge or control. Second, the disclosure of private data has become commonplace due to carelessness, theft, or legal actions.

Our research seeks to protect the privacy of *past, archived data* — such as copies of emails maintained by an email provider — against accidental, malicious, and legal attacks. Specifically, we wish to ensure that *all* copies of certain data become unreadable after a user-specified time, *without* any specific action on the part of a user, and even if an attacker obtains both a cached copy of that data and the user's cryptographic keys and passwords.

This paper presents *Vanish*, a system that meets this challenge through a novel integration of cryptographic techniques with global-scale, P2P, distributed hash tables (DHTs). We implemented a proof-of-concept *Vanish* prototype to use both the million-plus-node Vuze BitTorrent DHT and the restricted-membership OpenDHT. We evaluate experimentally and analytically the functionality, security, and performance properties of *Vanish*, demonstrating that it is practical to use and meets the privacy-preserving goals described above. We also describe two applications that we prototyped on *Vanish*: a Firefox plugin for Gmail and other Web sites and a *Vanishing File* application.

1 Introduction

We target the goal of creating data that self-destructs or vanishes *automatically* after it is no longer useful. Moreover, it should do so *without* any explicit action by the users or any party storing or archiving that data, in such a way that *all* copies of the data vanish simultaneously from all storage sites, online or offline.

Numerous applications could benefit from such self-destructing data. As one example, consider the case of email. Emails are frequently cached, stored, or archived by email providers (e.g., Gmail, or Hotmail), local backup systems, ISPs, etc. Such emails may cease to have value to the sender and receiver after a short period of time. Nevertheless, many of these emails are private, and the act of storing them indefinitely at intermediate locations creates a potential privacy risk. For example, imagine that Ann sends an email to her friend discussing a sensitive topic, such as her relationship with her husband, the possibility of a divorce, or how to ward off a spurious lawsuit (see Figure 1(a)). This email has no value as soon as her friend reads it, and Ann would like that *all* copies of this email — regardless of where stored or cached — be automatically destroyed after a certain period of time, rather than risk exposure in the future as part of a data breach, email provider mismanagement [41], or a legal action. In fact, Ann would prefer that these emails disappear early — and *not* be read by her friend — rather than risk disclosure to unintended parties. Both individuals and corporations could benefit from self-destructing emails.

More generally, self-destructing data is broadly applicable in today's Web-centered world, where users' sensitive data can persist "in the cloud" indefinitely (sometimes even after the user's account termination [61]). With self-destructing data, users can regain control over the lifetimes of their Web objects, such as private messages on Facebook, documents on Google Docs, or private photos on Flickr.

Numerous other applications could also benefit from self-destructing data. For example, while we do not condone their actions, the high-profile cases of several politicians [4, 62] highlight the relevance for self-destructing SMS and MMS text messages. The need for self-destructing text messages extends to the average user as well [42, 45]. As a news article states, "don't ever say anything on e-mail or text messaging that you don't want

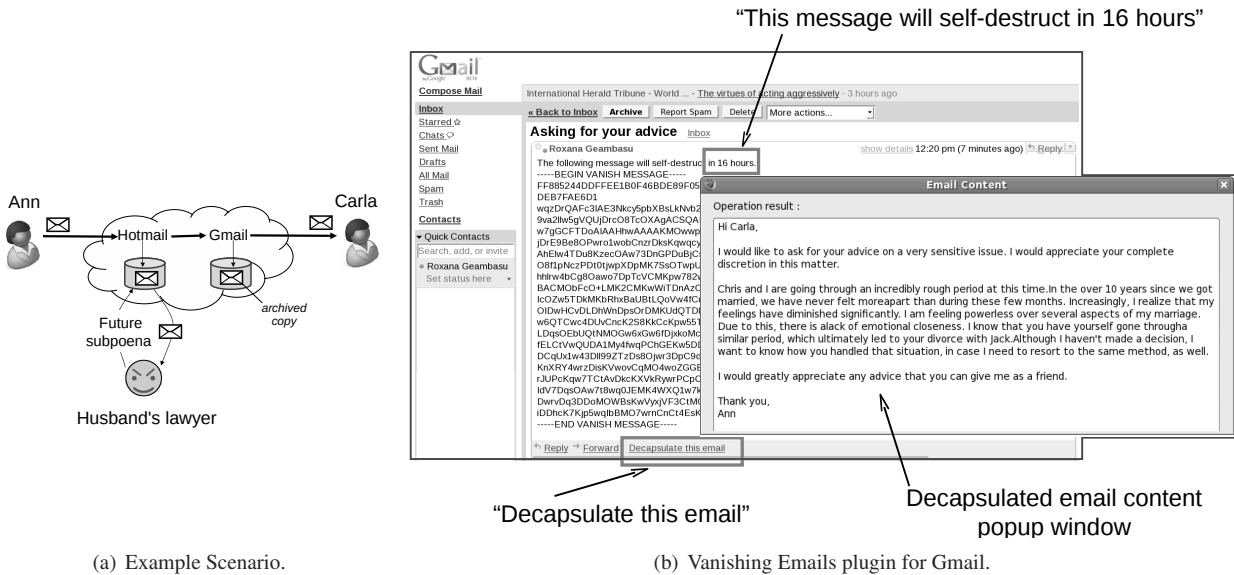


Figure 1: **Example Scenario and Vanish Email Screenshot.** (a) Ann wants to discuss her marital relationship with her friend, Carla, but does not want copies stored by intermediate services to be used in a potential child dispute trial in the future. (b) The screenshot shows how Carla reads a vanishing email that Ann has already sent to her using our Vanish Email Firefox plugin for Gmail.

to come back and bite you [42].” Some have even argued that the right and ability to destroy data are essential to protect fundamental societal goals like privacy and liberty [34, 44].

As yet another example, from a data sanitation perspective, many users would benefit from self-destructing trash bins on their desktops. These trash bins would preserve deleted files for a certain period of time, but after a timeout the files would self-destruct, becoming unavailable even to a forensic examiner (or anyone else, including the user). Moreover, the unavailability of these files would be guaranteed even if the forensic examiner is given a pristine copy of the hard drive from *before* the files self-destructed (e.g., because the machines were confiscated as part of a raid). Note that employing a whole disk encryption scheme is not sufficient, as the forensic examiner might be able to obtain the user’s encryption passwords and associated cryptographic keys through legal means. Other time-limited temporary files, like those that Microsoft Word periodically produces in order to recover from a crash [17], could similarly benefit from self-destructing mechanisms.

Observation and Goals. A key observation in these examples is that users need to keep certain data for only a limited period of time. After that time, access to that data should be revoked for *everyone* — including the legitimate users of that data, the known or unknown entities holding copies of it, and the attackers. This mechanism will not be universally applicable to all users or data types; instead, we focus in particular on sensitive data that a user would prefer to see destroyed early rather than fall into the wrong hands.

Motivated by the above examples, as well as our observation above, we ask whether it is possible to create a system that can permanently delete data after a timeout:

1. even if an attacker can retroactively obtain a pristine copy of that data *and* any relevant persistent cryptographic keys and passphrases from *before* that timeout, perhaps from stored or archived copies;
2. without the use of any explicit delete action by the user or the parties storing that data;
3. without needing to modify any of the stored or archived copies of that data;
4. without the use of secure hardware; and
5. without relying on the introduction of any *new* external services that would need to be deployed (whether trusted or not).

A system achieving these goals would be broadly applicable in the modern digital world as we’ve previously noted, e.g., for files, private blog posts, on-line documents, Facebook entries, content-sharing sites, emails, messages, etc. In fact, the privacy of any digital content could potentially be enhanced with self-deleting data.

However, implementing a system that achieves this goal set is challenging. Section 2 describes many natural approaches that one might attempt and how they all fall short. In this paper we focus on a specific self-deleting data scheme that we have implemented, using email as an example application.

Our Approach. The key insight behind our approach and the corresponding system, called *Vanish*, is to leverage the services provided by decentralized, global-scale P2P infrastructures and, in particular, Distributed Hash Tables (DHTs). As the name implies, DHTs are designed

to implement a robust index-value database on a collection of P2P nodes [64]. Intuitively, Vanish encrypts a user's data locally with a random encryption key not known to the user, *destroys* the local copy of the key, and then sprinkles bits (Shamir secret shares [49]) of the key across random indices (thus random nodes) in the DHT.

Our choice of DHTs as storage systems for Vanish stems from three unique DHT properties that make them attractive for our data destruction goals. First, their huge scale (over 1 million nodes for the Vuze DHT [28]), geographical distribution of nodes across many countries, and complete decentralization make them robust to powerful and legally influential adversaries. Second, DHTs are designed to provide reliable distributed storage [35, 56, 64]; we leverage this property to ensure that the protected data remains available to the user for a desired interval of time. Last but not least, DHTs have an inherent property that we leverage in a unique and non-standard way: the fact that the DHT is constantly changing means that the sprinkled information will naturally disappear (vanish) as the DHT nodes churn or internally cleanse themselves, thereby rendering the protected data permanently unavailable over time. In fact, it may be impossible to determine retroactively which nodes were responsible for storing a given value in the past.

Implementation and Evaluation. To demonstrate the viability of our approach, we implemented a proof-of-concept Vanish prototype, which is capable of using either Bittorrent's Vuze DHT client [3] or the PlanetLab-hosted OpenDHT [54]. The Vuze-based system can support 8-hour timeouts in the basic Vanish usage model and the OpenDHT-based system can support timeouts up to one week.¹ We built two applications on top of the Vanish core — a Firefox plugin for Gmail and other Web sites, and a self-destructing file management application — and we intend to distribute all of these as open source packages in the near future. While prototyping on existing DHT infrastructures not designed for our purpose has limitations, it allows us to experiment at scale, have users benefit immediately from our Vanish applications, and allow others to build upon the Vanish core. Figure 1(b) shows how a user can decapsulate a vanishing email from her friend using our Gmail plugin (complete explanation of the interface and interactions is provided in Section 5). Our performance evaluation shows that simple, Vanish-local optimizations can support even latency-sensitive applications, such as our Gmail plugin, with acceptable user-visible execution times.

Security is critical for our system and hence we consider it in depth. Vanish targets *post-facto*, *retroactive attacks*; that is, it defends the user against future attacks on

¹We have an external mechanism to extend Vuze timeouts beyond 8 hours, which we describe later.

old, forgotten, or unreachable copies of her data. For example, consider the subpoena of Ann's email conversation with her friend in the event of a divorce. In this context, the attacker does not know what specific content to attack until *after* that content has expired. As a result the attacker's job is very difficult, since he must develop an infrastructure capable of attacking *all* users at *all* times. We leverage this observation to estimate the cost for such an attacker, which we deem too high to justify a viable threat. While we target no formal security proofs, we evaluate the security of our system both analytically and experimentally. For our experimental attacks, we leverage Amazon's EC2 cloud service to create a Vuze deployment and to emulate attacks against medium-scale DHTs.

Contributions. While the basic idea of our approach is simple conceptually, care must be taken in handling and evaluating the mechanisms employed to ensure its security, practicality, and performance. Looking ahead, and after briefly considering other tempting approaches for creating self-destructing data (Section 2), the key contributions of this work are to:

- identify the principal requirements and goals for self-destructing data (Section 3);
- propose a novel method for achieving these goals that combines cryptography with decentralized, global-scale DHTs (Section 4);
- demonstrate that our prototype system and applications are deployable today using existing DHTs, while achieving acceptable performance, and examine the tensions between security and availability for such deployments (Section 5);
- experimentally and analytically evaluate the privacy-preservation capabilities of our DHT-based system (Section 6).

Together, these contributions provide the foundation for empowering users with greater control over the lifetimes of private data scattered across the Internet.

2 Candidate Approaches

A number of existing and seemingly natural approaches may appear applicable to achieving our objectives. Upon deeper investigation, however, we find that none of these approaches are sufficient to achieve the goals enumerated in Section 1. We consider these strawman approaches here and use them to further motivate our design constraints in Section 3.

The most obvious approach would require users to explicitly and manually delete their data or install a `cron` job to do that. However, because Web-mails and other Web data are stored, cached, or backed up at numerous places throughout the Internet or on Web servers,

this approach does not seem plausible. Even for a self-destructing trash bin, requiring the user to explicitly delete data is incompatible with our goals. For example, suppose that the hard disk fails and is returned for repairs or thrown out [15]; or imagine that a laptop is stolen and the thief uses a cold-boot [32] attack to recover its primary whole-disk decryption keys (if any). We wish to ensure data destruction even in cases such as these.

Another tempting approach might be to use a standard public key or symmetric encryption scheme, as provided by systems like PGP and its open source counterpart, GPG. However, traditional encryption schemes are insufficient for our goals, as they are designed to protect against adversaries without access to the decryption keys. Under our model, though, we assume that the attacker will be able to obtain access to the decryption keys, e.g., through a court order or subpoena.²

A potential alternative to standard encryption might be to use forward-secure encryption [6, 13], yet our goal is strictly stronger than forward secrecy. Forward secrecy means that if an attacker learns the state of the user's cryptographic keys at some point in time, they should not be able to decrypt data encrypted at an earlier time. However, due to caching, backup archives, and the threat of subpoenas or other court orders, we allow the attacker to either view past cryptographic state or force the user to decrypt his data, thereby violating the model for forward-secure encryption. For similar reasons, plus our desire to avoid introducing new trusted agents or secure hardware, we do not use other cryptographic approaches like key-insulated [5, 23] and intrusion-resilient [21, 22] cryptography. Finally, while exposure-resilient cryptography [11, 24, 25] allows an attacker to view parts of a key, we must allow an attacker to view all of the key.

Another approach might be to use steganography [48], deniable encryption [12], or a deniable file system [17]. The idea is that one could hide, deny the contents of, or deny the existence of private historical data, rather than destroying it. These approaches are also attractive but hard to scale and automate for many applications, e.g., generating plausible cover texts for emails and photos. In addition to the problems observed with deniable file systems in [17] and [38], deniable file systems would also create additional user hassles for a trash bin application, whereas our approach could be made invisible to the user.

For online, interactive communications systems, an ephemeral key exchange process can protect derived symmetric keys from future disclosures of asymmetric private keys. A system like OTR [1, 10] is particularly at-

tractive, but as the original OTR paper observes, this approach is not directly suited for less-interactive email applications, and similar arguments can be made for OTR's unsuitability for the other above-mentioned applications as well.

An approach with goals similar to ours (except for the goal of allowing users to create self-destructing objects without having to establish asymmetric keys or passphrases) is the Ephemerizer family of solutions [39, 46, 47]. These approaches require the introduction of one or more (possibly thresholded) trusted third parties which (informally) escrow information necessary to access the protected contents. These third parties destroy this extra data after a specified timeout. The biggest risks with such centralized solutions are that they may either not be trustworthy, or that even if they are trustworthy, users may still not trust them, hence limiting their adoption. Indeed, many users may be wary to the use of dedicated, centralized trusted third-party services after it was revealed that the Hushmail email encryption service was offering the cleartext contents of encrypted messages to the federal government [59]. This challenge calls for a decentralized approach with fewer real risks *and* perceived risks.

A second lesson can be learned from the Ephemerizer solutions in that, despite their introduction several years ago, these approaches have yet to see widespread adoption. This may in part be due to the perceived trust issues mentioned above, but an additional issue is that these solutions require the creation of new, supported and maintained services. We theorize that solutions that require *new* infrastructures have a greater barrier to adoption than solutions that can "parasitically" leverage *existing* infrastructures. A variant of this observation leads us to pursue approaches that do not require secure hardware or other dedicated services.

3 Goals and Assumptions

To support our target applications (self-destructing emails, Facebook messages, text messages, trash bins, etc.), we introduce the notion of a *vanishing data object* (VDO). A VDO encapsulates the user's data (such as a file or message) and prevents its contents from persisting indefinitely and becoming a source of retroactive information leakage. Regardless of whether the VDO is copied, transmitted, or stored in the Internet, it becomes unreadable after a predefined period of time even if an attacker *retroactively* obtains both a *pristine* copy of the VDO from before its expiration, and all of the user's past persistent cryptographic keys and passwords. Figure 2 illustrates the above properties of VDOs by showing the timeline for a typical usage of and attack against a VDO. We crystallize the assumptions underlying our

²U.S. courts are debating whether citizens are required to disclose private keys, although the ultimate verdict is unclear. We thus target technologies robust against a verdict in either direction [29, 40]. Other countries such as the U.K. [43] require release of keys, and coercion or force may be an issue in yet other countries.

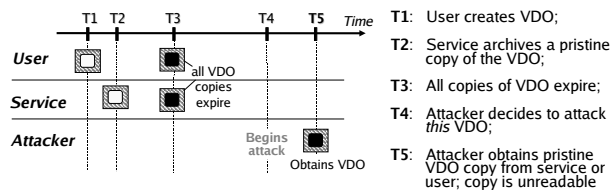


Figure 2: Timeline for VDO usage and attack.

VDO model and the central aspects of the threat model below.

Assumptions. Our VDO abstraction and Vanish system make several key assumptions:

1. *Time-limited value.* The VDO will be used to encapsulate data that is only of value to the user for a limited period of time.
2. *Known timeout.* When a user encapsulates data in a VDO, she knows the approximate lifetime that she wants for her VDO.
3. *Internet connectivity.* Users are connected to the Internet when interacting with VDOs.
4. *Dispensability under attack.* Rather than risk exposure to an adversary, the user prefers the VDO to be destroyed, even if prematurely.

We consider encapsulation of data that only needs to be available for hours or days; *e.g.*, certain emails, Web objects, SMSs, trash bin files, and others fall into this category. Internet connectivity is obviously required for many of our applications already, such as sending and receiving emails. More generally, the promise of ubiquitous connectivity makes this assumption reasonable for many other applications as well. Internet connectivity is not required for deletion, *i.e.*, a VDO will become unreadable even if connectivity is removed from its storage site (or if that storage site is offline). Finally, Vanish is designed for use with data that is private, but whose persistence is not critical. That is, while the user prefers that the data remain accessible until the specified timeout, its premature destruction is preferable to its disclosure.

Goals. Having stated these assumptions, we target the following functional goals and properties for Vanish:

1. *Destruction after timeout.* A VDO must expire automatically and *without* any explicit action on the part of its users or any party storing a copy of the VDO. Once expired, the VDO must also be inaccessible to any party who obtains a *pristine* copy of the VDO from *prior* to its expiration.
2. *Accessible until timeout.* During its lifetime, a VDO's contents should be available to legitimate users.
3. *Leverage existing infrastructures.* The system must leverage existing infrastructures. It must not rely on external, special-purpose dedicated services.

4. *No secure hardware.* The system must not require the use of dedicated secure hardware.
5. *No new privacy risks.* The system should not introduce new privacy risks to the users.

A corollary of goal (1) is that the VDO will become unavailable to the legitimate users after the timeout, which is compatible with our applications and assumption of time-limited value.

Our desire to leverage existing infrastructure (goal (3)) stems from our belief that special-purpose services may hinder adoption. As noted previously, Hushmail's disclosure of the contents of users' encrypted emails to the federal government [59] suggests that, even if the centralized service or a threshold subset of a collection of centralized services is trustworthy, users may still be unwilling to trust them.

As an example of goal (5), assume that Ann sends Carla an email *without* using Vanish, and then another email using Vanish. If an attacker cannot compromise the privacy of the first email, then we require that the same attacker — regardless of how powerful — cannot compromise the privacy of the second email.

In addition to these goals, we also seek to keep the VDO abstraction as generic as possible. In Vanish, the process of encapsulating data in a VDO does *not* require users to set or remember passwords or manage cryptographic keys. However, to ensure privacy under stronger threat models, Vanish applications may compose VDOs with traditional encryption systems like PGP and GPG. In this case, the user will naturally need to manipulate the PGP/GPG keys and passphrases.

Threat Models. The above list enumerates the intended properties of the system *without* the presence of an adversary. We now consider the various classes of potential adversaries against the Vanish system, as well as the desired behavior of our system in the presence of such adversaries.

The central security goal of Vanish is to ensure the destruction of data after a timeout, despite potential adversaries who might attempt to access that data after its timeout. Obviously, care must be taken in defining what a plausible adversary is, and we do that below and in Section 6. But we also stress that we explicitly do *not* seek to preserve goal (2) — accessible prior to a timeout — in the presence of adversaries. As previously noted, we believe that users would prefer to sacrifice availability pre-timeout in favor of assured destruction for the types of data we are protecting. For example, we do not defend against denial of service attacks that could prevent reading of the data during its lifetime. Making this assumption allows us to focus on the primary novel insights in this work: methods for leveraging decentralized, large-scale P2P networks in order to make data vanish over time.

We therefore focus our threat model and subsequent analyses on attackers who wish to compromise data privacy. Two key properties of our threat model are:

1. *Trusted data owners.* Users with legitimate access to the same VDOs trust each other.
2. *Retroactive attacks on privacy.* Attackers do not know which VDOs they wish to access until *after* the VDOs expire.

The former aspect of the threat model is straightforward, and in fact is a shared assumption with traditional encryption schemes: it would be impossible for our system to protect against a user who chooses to leak or permanently preserve the cleartext contents of a VDO-encapsulated file through out-of-band means. For example, if Ann sends Carla a VDO-encapsulated email, Ann must trust Carla not to print and store a hard-copy of the email in cleartext.

The latter aspect of the threat model — that the attacker does not know the identity of a specific VDO of interest until *after* its expiration — was discussed briefly in Section 1. For example, email or SMS subpoenas typically come long after the user sends a particular sensitive email. Therefore, our system defends the user against *future attacks against old copies of private data*.

Given the retroactive restriction, an adversary would have to do some precomputation prior to the VDO's expiration. The precise form of precomputation will depend on the adversary in question. The classes of adversaries we consider include: the user's employer, the user's ISP, the user's web mail provider, and unrelated malicious nodes on the Internet. For example, foreshadowing to Section 6, we consider an ISP that might spy on the connections a user makes to the Vuze DHT on the off chance that the ISP will later be asked to assist in the retroactive decapsulation of the user's VDO. Similarly, we consider the potential for an email service to proactively try to violate the privacy of VDOs prior to expiration, for the same reason. Although we deem both situations unlikely because of public perception issues and lack of incentives, respectively, we can also provide defenses against such adversaries.

Finally, we stress that we do not seek to provide privacy against an adversary who gets a warrant to intercept *future* emails. Indeed, such an attacker would have an arsenal of attack vectors at his disposal, including not only *a priori* access to sensitive emails but also keyloggers and other forensic tools [37].

4 The Vanish Architecture

We designed and implemented Vanish, a system capable of satisfying all of the goals listed in Section 3. A key contribution of our work is to leverage existing, decentralized, large-scale Distributed Hash Tables (DHTs).

After providing a brief overview of DHTs and introducing the insights that underlie our solution, we present our system's architecture and components.

Overview of DHTs. A DHT is a distributed, peer-to-peer (P2P) storage network consisting of multiple participating *nodes* [35, 56, 64]. The design of DHTs varies, but DHTs like Vuze generally exhibit a put/get interface for reading and storing data, which is implemented internally by three operations: *lookup*, *get*, and *store*. The data itself consists of an (*index*, *value*) pair. Each node in the DHT manages a part of an astronomically large index name space (e.g., 2^{160} values for Vuze). To store data, a client first performs a *lookup* to determine the nodes responsible for the index; it then issues a *store* to the responsible node, who saves that (*index*, *value*) pair in its local DHT database. To retrieve the value at a particular index, the client would *lookup* the nodes responsible for the index and then issue *get* requests to those nodes. Internally, a DHT may replicate data on multiple nodes to increase availability.

Numerous DHTs exist in the Internet, including Vuze, Mainline, and KAD. These DHTs are *communal*, i.e., any client can join, although DHTs such as OpenDHT [54] only allow authorized nodes to join.

Key DHT-related Insights. Three key properties of DHTs make them extremely appealing for use in the context of a self-destructing data system:

1. *Availability.* Years of research in availability in DHTs have resulted in relatively robust properties of today's systems, which typically provide good availability of data prior to a specific timeout. Timeouts vary, e.g., Vuze has a fixed 8-hour timeout, while OpenDHT allows clients to choose a per-data-item timeout of up to one week.
2. *Scale, geographic distribution, and decentralization.* Measurement studies of the Vuze and Mainline DHTs estimate in excess of one million simultaneously active nodes in each of the two networks [28]. The data in [63] shows that while the U.S. is the largest single contributor of nodes in Vuze, a majority of the nodes lie outside the U.S. and are distributed over 190 countries.
3. *Churn.* DHTs evolve naturally and dynamically over time as new nodes constantly join and old nodes leave. The average lifetime of a node in the DHT varies across networks and has been measured from minutes on Kazaa [30] to hours on Vuze/Azureus [28].

The first property provides us with solid grounds for implementing a useful system. The second property makes DHTs more resilient to certain types of attacks than centralized or small-scale systems. For example,

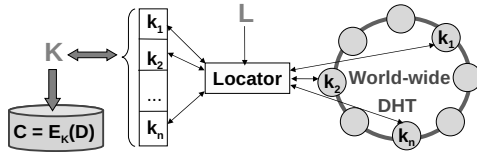


Figure 3: The Vanish system architecture.

while a centrally administered system can be compelled to release data by an attacker with legal leverage [59], obtaining subpoenas for multiple nodes storing a VDO's key pieces would be significantly harder, and in some cases impossible, due to their distribution under different administrative and political domains.

Traditionally, DHT research has tried to counter the negative effects of churn on availability. For our purposes, however, the constant churn in the DHT is an advantage, because it means that data stored in DHTs will naturally and irreversibly disappear over time as the DHT evolves. In many cases, trying to determine the contents of the DHT one week in the past — let alone several months or years — may be impossible, because many of the nodes storing DHT data will have left or changed their locations in the index space. For example, in Vuze, a node changes its location in the DHT whenever its IP address or port number changes, which typically happens periodically for dynamic IP addresses (*e.g.*, studies show that over 80% of the IPs change within 7 days [65]). This self-cleansing property of DHTs, coupled with its scale and global decentralization, makes them a felicitous choice for our self-destructing data system.

Vanish. Vanish is designed to leverage one or more DHTs. Figure 3 illustrates the high-level system architecture. At its core, Vanish takes a data object D (and possibly an explicit timeout T), and encapsulates it into a VDO V .

In more detail, to encapsulate the data D , Vanish picks a random data key, K , and encrypts D with K to obtain a ciphertext C . Not surprisingly, Vanish uses threshold secret sharing [58] to split the data key K into N pieces (*shares*) K_1, \dots, K_N . A parameter of the secret sharing is a *threshold* that can be set by the user or by an application using Vanish. The threshold determines how many of the N shares are required to reconstruct the original key. For example, if we split the key into $N = 20$ shares and the threshold is 10 keys, then we can compute the key given any 10 of the 20 shares. In this paper we often refer to the *threshold ratio* (or simply *threshold*) as the percentage of the N keys required, *e.g.*, in the example above the threshold ratio is 50%.

Once Vanish has computed the key shares, it picks at random an *access key*, L . It then uses a cryptographically secure pseudorandom number generator [7], keyed by L , to derive N indices into the DHT, I_1, \dots, I_N . Vanish then sprinkles the N shares K_1, \dots, K_N at these pseudorandom locations throughout the DHT; specifically, for each $i \in$

$\{1, \dots, N\}$, Vanish stores the share K_i at index I_i in the DHT. If the DHT allows a variable timeout, *e.g.*, with OpenDHT, Vanish will also set the user-chosen timeout T for each share. Once more than $(N - \text{threshold})$ shares are lost, the VDO becomes permanently unavailable.

The final VDO V consists of $(L, C, N, \text{threshold})$ and is sent over to the email server or stored in the file system upon encapsulation. The decapsulation of V happens in the natural way, assuming that it has not timed out. Given VDO V , Vanish (1) extracts the access key, L , (2) derives the locations of the shares of K , (3) retrieves the required number of shares as specified by the threshold, (4) reconstructs K , and (5) decrypts C to obtain D .

Threshold Secret Sharing, Security, and Robustness.

For security we rely on the property that the shares K_1, \dots, K_N will disappear from the DHT over time, thereby limiting a retroactive adversary's ability to obtain a sufficient number of shares, which must be \geq the threshold ratio. In general, we use a ratio of $< 100\%$, otherwise the loss of a single share would cause the loss of the key. DHTs do lose data due to churn, and therefore a smaller ratio is needed to provide robust storage prior to the timeout. We consider all of these issues in more detail later; despite the conceptual simplicity of our approach, significant care and experimental analyses must be taken to assess the durability of our use of large-scale, decentralized DHTs.

Extending the Lifetime of a VDO. For certain uses, the default timeout offered by Vuze might be too limiting. For such cases, Vanish provides a mechanism to refresh VDO shares in the DHT. While it may be tempting at first to simply use Vuze's republishing mechanism for index-value pairs, doing so would re-push the same pairs $(I_1, K_1), \dots, (I_N, K_N)$ periodically, until the timeout. This would, in effect, increase the exposure of those key shares to certain attackers. Hence, our refresh mechanism retrieves the original data key K before its timeout, re-splits it, obtaining a fresh set of shares, and derives new DHT indices I_1, \dots, I_N as a function of L and a weakly synchronized clock. The weakly synchronized clock splits UTC time into roughly 8-hour epochs and uses the epoch number as part of the input to the location function. Decapsulations then query locations generated from both the current epoch number and the neighboring epochs, thus allowing clocks to be weakly synchronized.

Naturally, refreshes require periodic Internet connectivity. A simple home-based setup, where a broadband connected PC serves as the user's refreshing proxy, is in our view and experience a very reasonable choice given today's highly connected, highly equipped homes. In fact, we have been using this setup in our in-house deployment of Vanish in order to achieve longer timeouts for our emails (see Section 5).

Using multiple or no DHTs. As an extension to the scheme above, it is possible to store the shares of the data key K in *multiple* DHTs. For example, one might first split K into two shares K' and K'' such that both shares are required to reconstruct K . K' is then split into N' shares and sprinkled in the Vuze DHT, while K'' is split into N'' shares and sprinkled in OpenDHT. Such an approach would allow us to argue about security under different threat models, using OpenDHT's closed access (albeit small scale) and Vuze's large scale (albeit communal) access.

An alternate model would be to abandon DHTs and to store the key shares on distributed but managed nodes. This approach bears limitations similar to Ephemerizer (Section 2). A hybrid approach might be to store shares of K' in a DHT and shares of K'' on managed nodes. This way, an attacker would have to subvert both the privately managed system *and* the DHT to compromise Vanish.

Forensic Trails. Although not a common feature in today's DHTs, a future DHT or managed storage system could additionally provide a forensic trail for monitoring accesses to protected content. A custom DHT could, for example, record the IP addresses of the clients that query for particular indices and make that information available to the originator of that content. The existence of such a forensic trail, even if probabilistic, could dissuade third parties from accessing the contents of VDOs that they obtain prior to timeout.

Composition. Our system is not designed to protect against all attacks, especially those for which solutions are already known. Rather, we designed both the system and our applications to be composable with other systems to support defense-in-depth. For example, our Vanish Gmail plugin can be composed with GPG in order to avoid VDO sniffing by malicious email services. Similarly, our system can compose with Tor to ensure anonymity and throttle targeted attacks.

5 Prototype System and Applications

We have implemented a Vanish prototype capable of integrating with both Vuze and OpenDHT. In this section, we demonstrate that (1) by leveraging existing, unmodified DHT deployments we can indeed achieve the core functions of vanishing data, (2) the resulting system supports a variety of applications, and (3) the performance of VDO operations is reasonable. We focus our discussions on Vuze because its large scale and dynamic nature make its analysis both more interesting and more challenging. A key observation derived from our study is a tension in setting VDO parameters (N and threshold) when targeting both high availability prior to the timeout and high security. We return to this tension in Section 6.

To integrate Vanish with the Vuze DHT, we made two

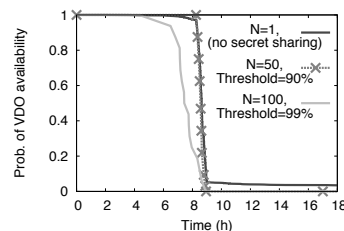


Figure 4: **VDO availability in the Vuze-based Vanish system.** The availability probability for single-key VDOs ($N = 1$) and for VDOs using secret sharing, averaged over 100 runs. Secret sharing is required to ensure pre-timeout availability and post-timeout destruction. Using $N = 50$ and a threshold of 90% achieves these goals.

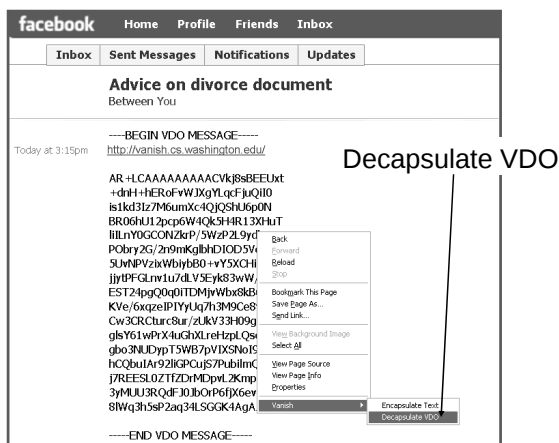
minor changes (< 50 lines of code) to the existing Vuze BitTorrent client: a security measure to prevent lookup sniffing attacks (see Section 6.2) and several optimizations suggested by prior work [28] to achieve reasonable performance for our applications. All these changes are *local* to Vanish nodes and do not require adoption by any other nodes in the Vuze DHT.

5.1 Vuze Background

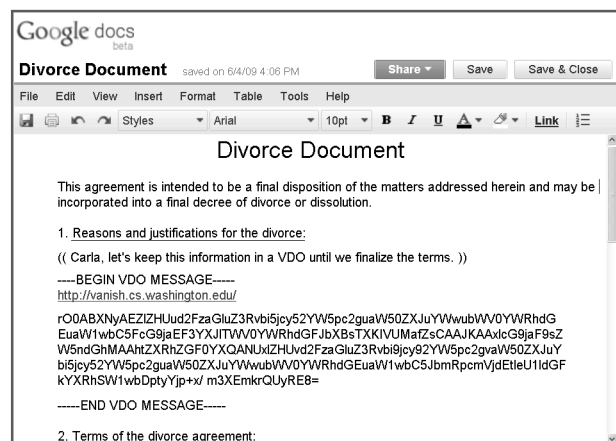
The Vuze (*a.k.a.* Azureus) DHT is based on the Kademlia [35] protocol. Each DHT node is assigned a “random” 160-bit ID based on its IP and port, which determines the index ranges that it will store. To store an (index, value) pair in the DHT, a client looks up 20 nodes with IDs closest to the specified index and then sends `store` messages to them. Vuze nodes republish the entries in their cache database every 30 minutes to the other 19 nodes closest to the value's index in order to combat churn in the DHT. Nodes further *remove* from their caches all values whose `store` timestamp is more than 8 hours old. This process has a 1-hour grace period. The originator node must re-push its 8-hour-old (index, value) pairs if it wishes to ensure their persistence past 8 hours.

5.2 VDO Availability and Expiration in Vuze

We ran experiments against the real global Vuze P2P network and evaluated the availability and expiration guarantees it provides. Our experiments pushed 1,000 VDO shares to pseudorandom indices in the Vuze DHT and then polled for them periodically. We repeated this experiment 100 times over a 3-day period in January 2009. Figure 4 shows the average probability that a VDO remains available as a function of the time since creation, for three different N and threshold values. For these experiments we used the standard 8-hour Vuze timeout (i.e., we did not use our refreshing proxy to re-push shares).



(a) Vanishing Facebook messages.



(b) Google Doc with vanishing parts.

Figure 5: **The Web-wide applicability of Vanish.** Screenshots of two example uses of vanishing data objects on the Web. (a) Carla is attempting to decapsulate a VDO she received from Ann in a Facebook message. (b) Ann and Carla are drafting Ann's divorce document using a Google Doc; they encapsulate sensitive, draft information inside VDOs until they finalize their position.

The $N = 1$ line shows the lifetime for a single share, which by definition does not involve secret sharing. The single-share VDO exhibits two problems: non-negligible probabilities for premature destruction ($\approx 1\%$ of the VDOs time out before 8 hours) and prolonged availability ($\approx 5\%$ of the VDOs continue to live long after 8 hours). The cause for the former effect is churn, which leads to early loss of the unique key for some VDOs. While the cause for the latter effect demands more investigation, we suspect that some of the single VDO keys are stored by DHT peers running non-default configurations. These observations suggest that the naive (one share) approach for storing the data key K in the DHT meets neither the availability nor the destruction goals of VDOs, thereby motivating our need for redundancy.

Secret sharing can solve the two lifetime problems seen with $N = 1$. Figure 4 shows that for VDOs with $N = 50$ and threshold of 90%, the probability of premature destruction and prolonged availability both become vanishingly small ($< 10^{-3}$). Other values for $N \geq 20$ achieve the same effect for thresholds of 90%. However, using very high threshold ratios leads to poor pre-timeout availability curves: e.g., $N = 100$ and a threshold of 99% leads to a VDO availability period of 4 hours because the loss of only two shares share makes the key unrecoverable. We will show in Section 6 that increasing the threshold increases security. Therefore, the choice of N and the threshold represents a tradeoff between security and availability. We will investigate this tradeoff further in Section 6.

5.3 Vanish Applications

We built two prototype applications that use a Vanish daemon running locally or remotely to ensure self-destruction of various types of data.

FireVanish. We implemented a Firefox plugin for the popular Gmail service that provides the option of sending and reading self-destructing emails. Our implementation requires no server-side changes. The plugin uses the Vanish daemon both to transform an email into a VDO before sending it to Gmail and similarly for extracting the contents of a VDO on the receiver side.

Our plugin is implemented as an extension of FireGPG (an existing GPG plugin for Gmail) and adds Vanish-related browser overlay controls and functions. Using our FireVanish plugin, a user types the body of her email into the Gmail text box as usual and then clicks on a "Create a Vanishing Email" button that the plugin overlays atop the Gmail interface. The plugin encapsulates the user's typed email body into a VDO by issuing a VDO-create request to Vanish, replaces the contents of the Gmail text box with an encoding of the VDO, and uploads the VDO email to Gmail for delivery. The user can optionally wrap the VDO in GPG for increased protection against malicious services. In our current implementation, each email is encapsulated with its own VDO, though a multi-email wrapping would also be possible (e.g., all emails in the same thread).

When the receiving user clicks on one of his emails, FireVanish inspects whether it is a VDO email, a PGP email, or a regular email. Regular emails require no further action. PGP emails are first decrypted and then inspected to determine whether the underlying message is a VDO email. For VDO emails, the plugin overlays a link "Decapsulate this email" atop Gmail's regular interface (shown previously in Figure 1(b)). Clicking on this link causes the plugin to invoke Vanish to attempt to retrieve the cleartext body from the VDO email. If the VDO has not yet timed out, then the plugin pops up a new window showing the email's cleartext body; otherwise, an error message is displayed.

FireVanish Extension for the Web. Self-destructing data is broadly applicable in today’s Web-oriented world, in which users often leave permanent traces on many Web sites [61]. Given the opportunity, many privacy-concerned users would likely prefer that certain messages on Facebook, documents on Google Docs, or instant messages on Google Talk disappear within a short period of time.

To make Vanish broadly accessible for Web usage, FireVanish provides a simple, generic, yet powerful, interface that permits all of these applications. Once the FireVanish plugin has been installed, a Firefox user can select text in any Web page input box, right click on that selected text, and cause FireVanish to replace that text *in-line* with an encapsulated VDO. Similarly, when reading a Web page containing a VDO, a user can select that VDO and right click to decapsulate it; in this case, FireVanish leaves the VDO in place and displays the cleartext in a separate popup window.

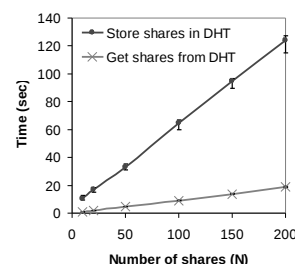
Figure 5 shows two uses of FireVanish to encapsulate and read VDOs within Facebook and Google Docs. The screenshots demonstrate a powerful concept: FireVanish can be used seamlessly to empower privacy-aware users with the ability to limit the lifetime of their data on Web applications that are unaware of Vanish.

Vanishing Files. Finally, we have implemented a vanishing file application, which can be used directly or by other applications, such as a self-destructing trash bin or Microsoft Word’s autosave. Users can wrap sensitive files into self-destructing VDOs, which expire after a given timeout. In our prototype, the application creates a VDO wrapping one or more files, deletes the cleartext files from disk, and stores the VDO in their place. This ensures that, even if an attacker copies the raw bits from the laptop’s disks after the timeout, the data within the VDO will be unavailable. Like traditional file encryption, Vanishing Files relies upon existing techniques for securely shredding data stored on disks or memory.

5.4 Performance Evaluation

We measured the performance of Vanish for our applications, focusing on the times to encapsulate and decapsulate a VDO. Our goals were to (1) identify the system’s performance bottlenecks and propose optimizations, and (2) determine whether our Vuze-based prototype is fast enough for our intended uses. Our measurements use an Intel T2500 DUO with 2GB of RAM, Java 1.6, and a broadband network.

To identify system bottlenecks, we executed VDO operations and measured the times spent in the three main runtime components: DHT operations (storing and getting shares), Shamir secret sharing operations (splitting/recomposing the data key), and encryp-



(a) Scalability of DHT operations.

N	Time (seconds)		
	Encapsulate VDO		Decapsulate VDO
	Without prepush	With prepush	
10	10.5	0.082	0.9
20	16.9	0.082	2.0
50	32.8	0.082	4.7
100	64.5	0.082	9.2
150	94.7	0.082	14.0
200	124.3	0.082	19.0

(b) VDO operation execution times.

Figure 6: Performance in the Vuze-based Vanish system. (a) The scalability of DHT operation times as a function of the number of shares being gotten from or stored in the DHT (results are averages over 20 trials and error bars indicate standard deviations). (b) Total VDO encapsulation (with and without pre-push) and decapsulation times for FireVanish for a 2KB email, $N = 50$, and threshold 90%.

tion/decryption. In general, the DHT component accounts for over 99% of the execution time for all Vanish operations on small and medium-size data (up to tens of MB, like most emails). For much larger data sizes (*e.g.*, files over hundreds of MB), the encryption/decryption becomes the dominating component.

Our experiments also revealed the importance of configuring Vuze’s parameters on our latency-aware applications. With no special tuning, Vuze took 4 minutes to store 50 shares, even using parallel stores. By employing several Vuze optimizations we lowered the 50-share store time by a factor of 7 (to 32 seconds). Our most effective optimization — significantly lowering Vuze’s UDP timeout based on suggestions from previous research [28] — proved non-trivial, though. In particular, as we deployed Vanish within our group, we learned that different Internet providers (*e.g.*, Qwest, Comcast) exhibited utterly different network behaviors and latencies, making the setting of any one efficient value for the timeout impossible. Hence, we implemented a control-loop-based mechanism by which Vanish automatically configures Vuze’s UDP timeout based on current network conditions. The optimization requires only node-local changes to Vuze.

Figure 6(a) shows how the optimized DHT operation times scale with the number of shares (N), for a fixed threshold of 90%, over a broadband connection (Comcast). Scaling with N is important in Vanish, as its se-

curity is highly dependent on this parameter. The graph shows that getting DHT shares are relatively fast — under 5 seconds for $N = 50$, which is reasonable for emails, trash bins, etc. The cost of storing VDO shares, however, can become quite large (about 30 seconds for $N = 50$), although it grows linearly with the number of shares. To mask the store delays from the user, we implemented a simple optimization, where Vanish proactively generates data keys and pre-pushes shares into the DHT. This optimization leads to an unnoticeable DHT encapsulation time of 82ms.

Combining the results in this section and Section 6, we believe that parameters of $N = 50$ and a threshold of 90% provide an excellent tradeoff of security and performance. With these parameters and the simple pre-push optimization we’ve described, user-visible latency for Vanish operations, such as creating or reading a Vanish email, is relatively low — just a few seconds for a 2KB email, as shown in Figure 6(b).

5.5 Anecdotal Experience with FireVanish

We have been using the FireVanish plugin within our group for several weeks. We also provided Vanish to several people outside of our group. Our preliminary experience has confirmed the practicality and convenience of FireVanish. We also learned a number of lessons even in this short period; for example, we found our minimalist interface to be relatively intuitive, even for a non-CS user to whom we gave the system, and the performance is quite acceptable, as we noted above.

We also identified several limitations in the current implementation, some that we solved and others that we will address in the future. For example, in the beginning we found it difficult to search for encrypted emails or data, since their content is encrypted and opaque to the Web site. For convenience, we modified FireVanish to allow users to construct emails or other data by mixing together non-sensitive cleartext blocks with self-destructing VDOs, as illustrated in Figure 5(b). This facilitates identifying information over and above the subject line. We did find that certain types of communications indeed require timeouts longer than 8 hours. Hence, we developed and used Vanish in a proxy setting, where a Vanish server runs on behalf of a user at an online location (e.g., the user’s home) and refreshes VDO shares as required to achieve each VDO’s intended timeout in 8-hour units. The user can then freely execute the Vanish plugin from any connection-intermittent location (e.g., a laptop).

We are planning an open-source release of the software in the near future and are confident that this release will teach us significantly more about the usability, limitations, and security of our system.

6 Security Analyses

To evaluate the security of Vanish, we seek to assess two key properties: that (1) Vanish does not introduce any *new* threats to privacy (goal (5) in Section 3), and (2) Vanish is secure against adversaries attempting to retroactively read a VDO post-expiration.

It is straightforward to see that Vanish adds no new privacy risks. In particular, the key shares stored in the DHT are *not* a function of the encapsulated data D ; only the VDO is a function of D . Hence, if an adversary is unable to learn D when the user does not use Vanish, then the adversary would be unable to learn D if the user does use Vanish. There are three caveats, however. First, external parties, like the DHT, might infer information about who is communicating with whom (although the use of an anonymization system like Tor can alleviate this concern). Second, given the properties of Vanish, users might choose to communicate information that they might not communicate otherwise, thus amplifying the consequences of any successful data breach. Third, the use of Vanish might raise new legal implications. In particular, the new “eDiscovery” rules embraced by the U.S. may require a user to preserve emails and other data once in anticipation of a litigious action. The exact legal implications to Vanish are unclear; the user might need to decapsulate and save any relevant VDOs to prevent them from automatic expiration.

We focus the remainder of this section on attacks targeted at retroactively revoking the privacy of data encapsulated within VDOs (this attack timeline was shown in Figure 2). We start with a broad treatment of such attacks and then dive deeply into attacks that integrate adversarial nodes directly into the DHT.

6.1 Avoiding Retroactive Privacy Attacks

Attackers. Our motivation is to protect against retroactive data disclosures, e.g., in response to a subpoena, court order, malicious compromise of archived data, or accidental data leakage. For some of these cases, such as the subpoena, the party initiating the subpoena is the obvious “attacker.” The final attacker could be a user’s ex-husband’s lawyer, an insurance company, or a prosecutor. But executing a subpoena is a complex process involving many other actors, including potentially: the user’s employer, the user’s ISP, the user’s email provider, unrelated nodes on the Internet, and other actors. For our purposes, we define *all* the involved actors as the “adversary.”

Attack Strategies. The architecture and standard properties of the DHT cause significant challenges to an adversary who does *not* perform any computation or data interception prior to beginning the attack. First, the key

shares are unlikely to remain in the DHT much after the timeout, so the adversary will be incapable of retrieving the shares directly from the DHT. Second, even if the adversary could legally subpoena the machines that hosted the shares in the past, the churn in Vuze makes it difficult to determine the identities of those machines; many of the hosting nodes would have long disappeared from the network or changed their DHT index. Finally, with Vuze nodes scattered throughout the globe [63], gaining legal access to those machines raises further challenges. In fact, these are all reasons why the use of a DHT such as Vuze for our application is compelling.

We therefore focus on what an attacker might do *prior* to the expiration of a VDO, with the goal of amplifying his ability to reveal the contents of the VDO in the *future*. We consider three principal strategies for such precomputation.

Strategy (1): Decapsulate VDO Prior to Expiration.

An attacker might try to obtain a copy of the VDO and revoke its privacy *prior* to its expiration. This strategy makes the most sense when we consider, e.g., an email provider that proactively decapsulates all VDO emails in real-time in order to assist in responding to future subpoenas. The natural defense would be to further encapsulate VDOs in traditional encryption schemes, like PGP or GPG, which we support with our FireVanish application. The use of PGP or GPG would prevent the web-mail provider from decapsulating the VDO prior to expiration. And, by the time the user is forced to furnish her PGP private keys, the VDO would have expired. For the self-destructing trash bin and the Vanishing Files application, however, the risk of this attack is minimal.

Strategy (2): Sniff User’s Internet Connection. An attacker might try to intercept and preserve the data users push into or retrieve from the DHT. An ISP or employer would be most appropriately positioned to exploit this vector. Two natural defenses exist for this: the first might be to use a DHT that by default encrypts communications between nodes. Adding a sufficient level of encryption to existing DHTs would be technically straightforward assuming that the ISP or employer were passive and hence not expected to mount man-in-the-middle attacks. For the encryption, Vanish could compose with an ephemeral key exchange system in order to ensure that these encrypted communications remain private even if users’ keys are later exposed. Without modifying the DHT, the most natural solution is to compose with Tor [19] to tunnel one’s interactions with a DHT through remote machines. One could also use a different exit node for each share to counter potentially malicious Tor exit nodes [36, 66], or use Tor for only a subset of the shares.

Strategy (3): Integrate into DHT. An attacker might try

to integrate itself into the DHT in order to: create copies of all data that it is asked to store; intercept internal DHT lookup procedures and then issue `get` requests of his own for learned indices; mount a Sybil attack [26] (perhaps as part of one of the other attacks); or mount an Eclipse attack [60]. Such DHT-integrated attacks deserve further investigation, and we provide such an analysis in Section 6.2.

We will show from our experiments in Section 6.2 that an adversary would need to join the 1M-node Vuze DHT with approximately 80,000–90,000 malicious nodes to mount a store-based attack and capture a reasonable percentage of the VDOs (e.g., 25%). Even if possible, sustaining such an attack for an extended period of time would be prohibitively expensive (close to \$860K/year in Amazon EC2 computation and networking costs). The lookup-based attacks are easy to defeat using localized changes to Vanish clients. The Vuze DHT already includes rudimentary defenses against the Sybil attack and a full deployment of Vanish could leverage the existing body of works focused on hardening DHTs against Sybil and Eclipse attacks [9, 14, 16, 26, 51].

Deployment Decisions. Given attack strategies (1) and (2), a user of FireVanish, Vanishing Files, or any future Vanish-based application is faced with several options: to use the basic Vanish system or to compose Vanish with other security mechanisms like PGP/GPG or Tor. The specific decision is based on the threats to the user for the application in question.

Vanish is oriented towards personal users concerned that old emails, Facebook messages, text messages, or files might come back to “bite” them, as eloquently put in [42]. Under such a scenario, an ISP trying to assist in future subpoenas seems unlikely, thus we argue that composing Vanish with Tor is unnecessary for most users. The use of Tor seems even less necessary for some of the threats we mentioned earlier, like a thief with a stolen laptop.

Similarly, it is reasonable to assume that email providers will not proactively decapsulate and archive Vanishing Emails prior to expiration. One factor is the potential illegality of such accesses under the DMCA, but even without the DMCA this seems unlikely. Therefore, users can simply employ the FireVanish Gmail plugin without needing to exchange public keys with their correspondents. However, because our plugin extends FireGPG, any user already familiar with GPG could leverage our plugin’s GPG integration.

Data Sanitization. In addition to ensuring that Vanish meets its security and privacy goals, we must verify that the surrounding operating environment does not preserve information in a non-self-destructing way. For this reason, the system could leverage a broad set of ap-

proaches for sanitizing the Vanish environment, including secure methods for overwriting data on disk [31], encrypting virtual memory [50], and leveraging OS support for secure deallocation [15]. However, even absent those approaches, forensic analysis would be difficult if attempted much later than the data's expiration for the reasons we've previously discussed: by the time the forensic analysis is attempted relevant data is likely to have disappeared from the user's machine, the churn in the DHT would have made shares (and nodes) vanish irrevocably.

6.2 Privacy Against DHT-Integrated Adversaries

We now examine whether an adversary who interacts with the DHT *prior* to a VDO's expiration can, in the future, aid in retroactive attacks against the VDO's privacy. During such a precomputation phase, however, the attacker does not know which VDOs (or even which users) he might eventually wish to attack. While the attacker could compile a list of worthwhile targets (e.g., politicians, actors, etc.), the use of Tor would thwart such targeted attacks. Hence, the principle strategy for the attacker would be to create a copy of as many key shares as possible. Moreover, the attacker must do this continuously — 24x7 — thereby further amplifying the burden on the attacker.

Such an attacker might be *external* to the DHT — simply using the standard DHT interface in order to obtain key shares — or *internal* to the DHT. While the former may be the only available approach for DHTs like OpenDHT, the approach is also the most limiting to an attacker since the shares are stored at pseudorandomly generated and hence unpredictable indices. An attacker integrating into a DHT like Vuze has significantly more opportunities and we therefore focus on such DHT-integrating adversaries here.

Experimental Methodology. We ran extensive experiments on a private deployment of the Vuze DHT. In each experiment, a set of honest nodes pushed VDO shares into the DHT and retrieved them at random intervals of time, while malicious nodes sniffed *stores* and *lookups*.³ Creating our own Vuze deployment allowed us to experiment with various system parameters and workloads that we would not otherwise have been able to manipulate. Additionally, experimenting with attacks against Vuze at sufficient scale would have been prohibitively costly for us, just as it would for an attacker.

Our experiments used 1,000, 2,000, 4,500, and 8,000-node DHTs, which are significantly larger than those used for previous empirical DHT studies (e.g. 1,000

nodes in [53]). For the 8,000-node experiments we used 200 machine instances of Amazon's EC2 [2] compute cloud. For smaller experiments we used 100 of Emulab's 3GHz, 2GB machines [27]. In general, memory is the bottleneck, as each Vuze node must run in a separate process to act as a distinct DHT node. Approximately 50 Vuze nodes fit on a 2-GB machine.

Churn (node death and birth) is modeled by a Poisson distribution as in [53]. Measurements of DHT networks have observed different median lifetime distributions, e.g., 2.4 minutes for Kazaa [30], 60 minutes for Gnutella [57], and 5 hours with Vuze [28] (although this measurement may be biased towards longer-lived nodes). We believe that these vast differences stem from different content and application types that rely on these networks (e.g., the difference between audio and video clips). We chose a 2-hour median node lifetime, which provides insight into the availability—security tradeoffs under high churn.

6.2.1 The Store Sniffing Attack

We first examine a *store* sniffing attack in which the adversary saves all of the index-to-value mappings it receives from peers via *store* messages. Such an attacker might receive a VDO's key shares in one of two ways: directly from the user during a VDO's creation or refresh, or via replication. In Vuze, nodes replicate their cached index-to-value mappings every 30 minutes by pushing each mapping to 20 nodes whose IDs are closest to the mapping's index.

Effects of VDO Parameters on Security. Our first goal is to assess how security is affected by the VDO parameters N (the number of key shares distributed for each VDO) and the key threshold (the percent of the N shares required to decrypt a VDO). Figure 7(a) plots the probability that an attacker can capture sufficient key shares to revoke the privacy of a given VDO as a function of N and the threshold. This figure assumes the attacker has compromised 5% of the nodes in a 1,000-node DHT. Not surprisingly, as the number of shares N increases, the attacker's success probability drops significantly. Similarly, increasing the threshold increases security (i.e., decreases the attacker's success probability).

Availability is also affected by the VDO parameters and the tradeoff is shown in Figure 7(b). Here we see the maximum timeout (i.e., the VDO's lifetime) as a function of N and the threshold. The maximum VDO timeout is the largest time at which 99% of a set of 1,000 VDOs remained available in our experiment. The timeout is capped by our 10-hour experimental limit. From the figure, we see that increasing N improves not only security, but also availability. We also see that smaller thresholds support longer timeouts, because the system can toler-

³Vuze *get* messages do not reveal additional information about values stored in the DHT, so we do not consider them.

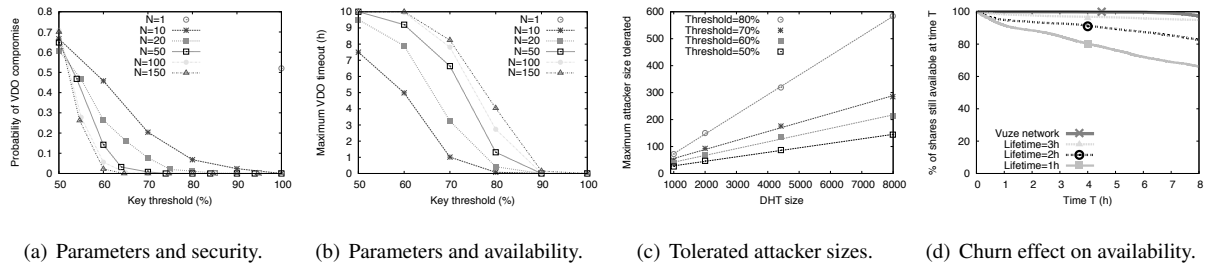


Figure 7: Analysis of the store sniffing attack. Fig. (a): the attacker’s success probability with increasing N and key threshold for a 1000-node DHT with 50 malicious nodes. Larger N and high thresholds ($\geq 65\%$) provide good security. Fig. (b): maximum VDO timeout supported for a .99 availability level. Large N with smaller key thresholds ($\leq 70\%$) provide useful VDO timeouts. Fig. (c): maximum number of attacker nodes that a DHT can tolerate, while none of the 1,000 VDOs we pushed were compromised. Fig. (a), (b), and (c) assume 2-hour churn. Fig. (d): the single-share availability decreases over time for different churn models in our private network and for the real Vuze network.

ate more share loss. The choice of threshold thus involves a tradeoff between security and availability: high thresholds provide more security and low thresholds provide longer lifetime. For example, if a lifetime of only 4 hours is needed — which might be reasonable for certain emails or SMSs — then choosing $N = 50$ and threshold 75% leads to good security and performance. If a timeout of 8 hours is required, $N = 100$ and threshold of 70% is a good tradeoff for the 2-hour churn. Thus, by tuning N and the key threshold, we can obtain high security, good availability, and reasonable performance in the context of a small 1,000-node DHT and 5% attackers.

Attacker Sizes. We now consider how many attacker nodes a DHT deployment of a given size can tolerate with small chance that the attacker succeeds in pre-obtaining a sufficient number of shares for any VDO. Figure 7(c) shows the maximum attacker sizes tolerated by DHTs of increasing sizes, for various key thresholds. The values are calculated so as to ensure that none of the 1,000 VDOs we experimented with was compromised. We computed these values from experiments using $N = 150$, 2-hour churn, and various attacker sizes for each DHT size. For an 8,000-node DHT, even if 600 nodes are controlled by a store-sniffing attacker, the adversary would still not obtain any of our 1,000 VDOs.

More important, Figure 7(c) suggests that the number of attackers that the DHT can tolerate grows linearly with DHT size. Assuming this trend continues further, we estimate that, in a 1M-node DHT, an attacker with 35,000 nodes would still have less than 10^{-3} probability of recording a sufficient number of shares to compromise a single VDO with $N = 150$ and a threshold of 70%.

We have also experimented with a different metric of success: requiring an attacker to obtain enough key shares to compromise at least 25% of all VDOs. Concretely, for $N = 150$ and a threshold of 80%, our experiment with a 8,000 node DHT required the attacker to control over 710 nodes. This value also appears to grow

linearly in the size of the DHT; extrapolating to a 1M-node DHT, such an attack would require at least 80,000 malicious nodes. We believe that inserting this number of nodes into the DHT, while possible for limited amounts of time, is too expensive to do continuously (we provide a cost estimate below).

Finally, we note that our refresh mechanism for extending Vuze timeouts (explained in Section 4) provides good security properties in the context of store sniffing attacks. Given that our mechanism pushes new shares in each epoch, an attacker who fails to capture sufficient shares in one epoch must start anew in the next epoch and garner the required threshold from zero.

Setting Parameters for the Vuze Network. These results provide a detailed study of the store sniffing attack in the context of a 2-hour churn model induced on a private Vuze network. We also ran a selected set of similar availability and store attack experiments against a private network with a 3-hour churn model, closer to what has been measured for Vuze.⁴ The resulting availability curve for the 3-hour churn now closely resembles the one in the real Vuze network (see Figure 7(d)). In particular, for both the real network and the private network with a 3-hour churn model, a ratio of 90% and $N \geq 20$ are enough to ensure VDO availability of 7 hours with .99 probability. Thus, from an availability standpoint, the longer lifetimes allow us to raise the threshold to 90% to increase security.

From a security perspective, our experiments show that for an 8,000-node DHT, 3-hour churn model, and VDOs using $N = 50$ and threshold 90%, the attacker requires at least 820 nodes in order to obtain $\geq 25\%$ of the VDOs. This extrapolates to a requirement of $\approx 87,000$ nodes on Vuze to ensure attack effectiveness. Returning to our cost argument, while cloud computing in a system such as Amazon EC2 is generally deemed in-

⁴We used VDOs of $N = 50$ and thresholds of 90% for these experiments.

expensive [18], the cost to mount a year-long 87,000-node attack would be over \$860K for processing and Internet traffic alone, which is sufficiently high to thwart an adversary's compromise plans in the context of our personal use targets (*e.g.*, seeking sensitive advice from friends over email). Of course, for larger N (*e.g.*, 150), an attacker would be required to integrate even more nodes and at higher cost. Similarly, the cost of an attack would increase as more users join the Vuze network.

Overall, to achieve good performance, security and availability, we recommend using $N = 50$ and a threshold of 90% for VDOs in the current Vuze network. Based on our experiments, we conclude that under these parameters, an attacker would be required to compromise between 8—9% of the Vuze network in order to be effective in his attack.

6.2.2 The Lookup Sniffing Attack

In addition to seeing `store` requests, a DHT-integrated adversary also sees `lookup` requests. Although Vuze only issues lookups prior to `storing` and `getting` data objects, the lookups pass through multiple nodes and hence provide additional exposure for VDO key shares. In a lookup sniffing attack, whenever an attacker node receives a lookup for an index, it actively fetches the value stored at that index, if any. While more difficult to handle than the passive `store` attack, the `lookup` attack could increase the adversary's effectiveness.

Fortunately, a simple, *node-local* change to the Vuze DHT thwarts this attack. Whenever a Vanish node wants to store to or retrieve a value from an index I , the node looks up an *obfuscated* index I' , where I' is related to but different from I . The client then issues a `store/get` for the original index I to the nodes returned in response to the lookup for I' . In this way, the retrieving node greatly reduces the number of other nodes (and potential attackers) who see the real index.

One requirement governs our simple choice of an obfuscation function: the same set of replicas must be responsible for both indexes I and I' . Given that Vuze has 1M nodes and that IDs are uniformly distributed (they are obtained via hashing), all mappings stored at a certain node should share approximately the higher-order $\log_2(10^6) \approx 20$ bits with the IDs of the node. Thus, looking up only the first 20b of the 160b of a Vuze index is enough to ensure that the nodes resulted from the lookup are indeed those in charge of the index. The rest of the index bits are useless in lookups and can be randomized, and are rehabilitated only upon sending the final `get/store` to the relevant node(s). We conservatively choose to randomize the last 80b from every index looked up while retrieving or storing mappings.

Lacking full index information, the attacker would

have to try retrieving all of the possible indexes starting with the obfuscated index (2^{80} indexes), which is impossible in a timely manner. This Vuze change was trivial (only 10 lines of modified code) and it is completely local to Vanish nodes. That is, the change does not require adoption by any other nodes in the DHT to be effective.

6.2.3 Standard DHT Attacks

In the previous sections we offered an in-depth analysis of two data confidentiality attacks in DHTs (store and lookup sniffing), which are specific in the context of our system. However, the robustness of communal DHTs to more general attacks has been studied profusely in the past and such analyses, proposed defenses, and limitations are relevant to Vanish, as well. Two main types of attacks identified by previous works are the Sybil attack [26] and the Eclipse (or route hijacking) attack [60]. In the Sybil attack, a few malicious nodes assume a large number of identities in the DHT. In the Eclipse attack, several adversarial nodes can redirect most of the traffic issued by honest nodes toward themselves by poisoning their routing tables with malicious node contact information [60].

The Vuze DHT already includes a rudimentary defense against Sybil attacks by constraining the identity of a Vuze node to a function of its IP address and port modulo 1999. While this measure might be sufficient for the early stages of a Vanish deployment, stronger defenses are known, *e.g.*, certified identities [26] and periodic cryptographic puzzles [9] for defense against Sybil attacks and various other defenses against Eclipse attacks [14, 51]. Given that the core Vanish system is network-agnostic, we could easily port our system onto more robust DHTs implementing stronger defenses. Moreover, if Vanish-style systems become popular, it would also be possible to consider Vanish-specific defenses that could leverage, *e.g.*, the aforementioned tight coupling between Vanish and the identities provided by PGP public keys. Finally, while we have focused on the Vuze DHT — and indeed its communal model makes analyzing security more interesting and challenging — Vanish could also split keys across *multiple* DHTs, or even DHTs and managed systems, as previously noted (Section 4). The different trust models, properties, and risks in those systems would present the attacker with a much more difficult task.

7 Related Work

We have discussed a large amount of related work in Section 2 and throughout the text. As additional related work, the Adeona system also leverages DHTs for increased privacy, albeit with significantly different goals [55]. Several existing companies aim to achieve

similar goals to ours (e.g., self-destructing emails), but with very different threat models (company servers must be trusted) [20]. Incidents with Hushmail, however, may lead users to question such trust models [59]. There also exists research aimed at destroying archived data where the data owner has the ability to explicitly and manually erase extra data maintained elsewhere, e.g., [8]; we avoid such processes, which may not always succeed or may be vulnerable to their own accidental copying or disclosures. Finally, albeit with different goals and perspectives, Rabin proposes an information-theoretically secure encryption system that leverages a decentralized collection of dedicated machines that continuously serve random pages of data [52], which is related to the limited storage model [33]. Communicants, who pre-share symmetric keys, can download and xor specific pages together to derive a one-time pad. The commonality between our approach and Rabin's is in the use of external machines to assist in privacy; the model, reliance on dedicated services, and pre-negotiation of symmetric keys between communicants are among the central differences.

8 Conclusions

Data privacy has become increasingly important in our litigious and online society. This paper introduced a new approach for protecting data privacy from attackers who retroactively obtain, through legal or other means, a user's stored data and private decryption keys. A novel aspect of our approach is the leveraging of the essential properties of modern P2P systems, including churn, complete decentralization, and global distribution under different administrative and political domains. We demonstrated the feasibility of our approach by presenting *Vanish*, a proof-of-concept prototype based on the Vuze global-scale DHT. *Vanish* causes sensitive information, such as emails, files, or text messages, to irreversibly self-destruct, without any action on the user's part and without any centralized or trusted system. Our measurement and experimental security analysis sheds insight into the robustness of our approach to adversarial attacks.

Our experience also reveals limitations of existing DHTs for *Vanish*-like applications. In Vuze, for example, the fixed data timeout and large replication factor present challenges for a self-destructing data system. Therefore, one exciting direction of future research is to redesign existing DHTs with our specific privacy applications in mind. Our plan to release the current *Vanish* system will help to provide us with further valuable experience to inform future DHT designs for privacy applications.

9 Acknowledgements

We offer special thanks to Steve Gribble, Arvind Krishnamurthy, Mark McGovern, Paul Ohm, Michael Piatek, and our anonymous reviewers for their comments on the paper. This work was supported by NSF grants NSF-0846065, NSF-0627367, and NSF-614975, an Alfred P. Sloan Research Fellowship, the Wissner-Slivka Chair, and a gift from Intel Corporation.

References

- [1] C. Alexander and I. Goldberg. Improved user authentication in off-the-record messaging. In *WPES*, 2007.
- [2] Amazon.com. Amazon elastic compute cloud (EC2). <http://aws.amazon.com/ec2/>, 2008.
- [3] Azureus. <http://www.vuze.com/>.
- [4] BBC News. US mayor charged in SMS scandal. <http://news.bbc.co.uk/2/hi/americas/7311625.stm>, 2008.
- [5] M. Bellare and A. Palacio. Protecting against key exposure: Strongly key-insulated encryption with optimal threshold. *Applicable Algebra in Engineering, Communication and Computing*, 16(6), 2006.
- [6] M. Bellare and B. Yee. Forward security in private key cryptography. In M. Joye, editor, *CT-RSA 2003*, 2003.
- [7] M. Blum and S. Micali. How to generate cryptographically strong sequences of pseudo-random bits. In *Proceedings of the 23rd IEEE Symposium on Foundations of Computer Science (FOCS '82)*, 1982.
- [8] D. Boneh and R. Lipton. A revocable backup system. In *USENIX Security*, 1996.
- [9] N. Borisov. Computational puzzles as Sybil defenses. In *Proc. of the Intl. Conference on Peer-to-Peer Computing*, 2006.
- [10] N. Borisov, I. Goldberg, and E. Brewer. Off-the-record communication, or, why not to use PGP. In *WPES*, 2004.
- [11] R. Canetti, Y. Dodis, S. Halevi, E. Kushilevitz, and A. Sahai. Exposure-resilient functions and all-or-nothing transforms. In B. Preneel, editor, *EUROCRYPT 2000*, volume 1807 of *LNCs*, pages 453–469, Bruges, Belgium, May 14–18, 2000. Springer-Verlag, Berlin, Germany.
- [12] R. Canetti, C. Dwork, M. Naor, and R. Ostrovsky. Deniable encryption. In B. S. K. Jr., editor, *CRYPTO '97*, 1997.
- [13] R. Canetti, S. Halevi, and J. Katz. A forward-secure public-key encryption scheme. In *EUROCRYPT 2003*, 2003.
- [14] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. S. Wallach. Secure routing for structured peer-to-peer overlay networks. In *Proc. of OSDI*, 2002.
- [15] J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum. Shredding your garbage: Reducing data lifetime through secure deallocation. In *USENIX Security*, 2005.
- [16] T. Condie, V. Kacholia, S. Sankararaman, J. M. Hellerstein, and P. Maniatis. Induced churn as shelter from routing table poisoning. In *Proc. of NDSS*, 2006.
- [17] A. Czeskis, D. J. S. Hilaire, K. Koscher, S. D. Gribble, T. Kohno, and B. Schneier. Defeating encrypted and deniable file systems: TrueCrypt v5.1a and the case of the tattling OS and applications. In *3rd USENIX HotSec*, July 2008.
- [18] M. Dama. Amazon EC2 scalable processing power. <http://www.maxdama.com/2008/08/amazon-ec2-scalable-processing-power.html>, 2008.
- [19] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *USENIX Security*, 2004.
- [20] Disappearing Inc. Disappearing Inc. product page. <http://www.specimenbox.com/di/ab/hwdi.html>, 1999.

- [21] Y. Dodis, M. K. Franklin, J. Katz, A. Miyaji, and M. Yung. Intrusion-resilient public-key encryption. In *CT-RSA 2003*, volume 2612, pages 19–32. Springer-Verlag, Berlin, Germany, 2003.
- [22] Y. Dodis, M. K. Franklin, J. Katz, A. Miyaji, and M. Yung. A generic construction for intrusion-resilient public-key encryption. In T. Okamoto, editor, *CT-RSA 2004*, volume 2964 of *LNCS*, pages 81–98, San Francisco, CA, USA, Feb. 23–27, 2004. Springer-Verlag, Berlin, Germany.
- [23] Y. Dodis, J. Katz, S. Xu, and M. Yung. Key-insulated public key cryptosystems. In *EUROCRYPT 2002*, 2002.
- [24] Y. Dodis, A. Sahai, and A. Smith. On perfect and adaptive security in exposure-resilient cryptography. In *EUROCRYPT 2001*, volume 2045 of *LNCS*, pages 301–324. Springer-Verlag, Berlin, Germany, 2001.
- [25] Y. Dodis and M. Yung. Exposure-resilience for free: The case of hierarchical ID-based encryption. In *IEEE International Security In Storage Workshop*, 2002.
- [26] J. R. Douceur. The sybil attack. In *International Workshop on Peer-to-Peer Systems*, 2002.
- [27] Emulab. Emulab – network emulation testbed. <http://www.emulab.net/>, 2008.
- [28] J. Falkner, M. Piatek, J. John, A. Krishnamurthy, and T. Anderson. Profiling a million user DHT. In *Internet Measurement Conference*, 2007.
- [29] D. Goodin. Your personal data just got permanently cached at the US border. http://www.theregister.co.uk/2008/05/01/electronic_searches_at_us_borders/, 2008.
- [30] K. P. Gummadi, R. J. Dunn, S. Saroiu, S. D. Gribble, H. M. Levy, and J. Zahorjan. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *Proc. of SOSP*, 2003.
- [31] P. Gutmann. Secure deletion of data from magnetic and solid-state memory. In *USENIX Security*, 1996.
- [32] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: Cold boot attacks on encryption keys. In *USENIX Security*, 2008.
- [33] U. M. Maurer. Conditionally-perfect secrecy and a provably-secure randomized cipher. *Journal of Cryptology*, 5:53–66, 1992.
- [34] V. Mayer-Schoenberger. Useful Void: the art of forgetting in the age of ubiquitous computing. *Working Paper, John F. Kennedy School of Government, Harvard University*, 2007.
- [35] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proc. of Peer-to-Peer Systems*, 2002.
- [36] D. McCoy, K. Bauer, D. Grunwald, T. Kohno, and D. Sicker. Shining light in dark places: Understanding the Tor network. In *Privacy Enhancing Technologies Symposium*, July 2008.
- [37] D. McCullagh. Feds use keylogger to thwart PGP, Hushmail. news.cnet.com/8301-10784_3-9741357-7.html, 2008.
- [38] D. McCullagh. Security guide to customs-proofing your laptop. http://www.news.com/8301-13578_3-9892897-38.html, 2008.
- [39] S. K. Nair, M. T. Dashti, B. Crispo, and A. S. Tanenbaum. A hybrid PKI-IBC based ephemizer system. In *International Information Security Conference*, 2007.
- [40] E. Nakashima. Clarity sought on electronic searches. <http://www.washingtonpost.com/wp-dyn/content/article/2008/02/06/AR2008020604763.html>, 2008.
- [41] New York Times. F.B.I. Gained Unauthorized Access to E-Mail. http://www.nytimes.com/2008/02/17/washington/17fisa.html?_r=1&hp=&adxnnl=1&oref=slogin&adxnnlx=1203255399-44ri626iqXg7QNmwzoerKa, 2008.
- [42] News 24. Think before you SMS. <http://www.news24.com/News24/Technology/News/0,,2-13-1443-1541201,00.html>, 2004.
- [43] Office of Public Sector Information. Regulation of Investigatory Powers Act (RIPA), Part III – Investigation of Electronic Data Protected by Encryption etc. http://www.opsi.gov.uk/acts/acts2000/ukpga_20000023-en-8,2000.
- [44] P. Ohm. The Fourth Amendment right to delete. *The Harvard Law Review*, 2005.
- [45] PC Magazine. Messages can be forever. <http://www.pcmag.com/article2/0,1759,1634544,00.asp>, 2004.
- [46] R. Perlman. The Ephemizer: Making data disappear. *Journal of Information System Security*, 1(1), 2005.
- [47] R. Perlman. File system design with assured delete. In *Security in Storage Workshop (SISW)*, 2005.
- [48] F. A. P. Petitcolas, R. J. Anderson, and M. G. Kuhn. Information hiding: A survey. *Proceedings of the IEEE*, 87(7), 1999.
- [49] B. Poettering. "ssss: Shamir's Secret Sharing Scheme". <http://point-at-infinity.org/ssss/>, 2006.
- [50] N. Provos. Encrypting virtual memory. In *USENIX Security*, 2000.
- [51] K. P. N. Puttaswamy, H. Zheng, and B. Y. Zhao. Securing structured overlays against identity attacks. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2008.
- [52] M. O. Rabin. Provably unbreakable hyper-encryption in the limited access model. In *IEEE Information Theory Workshop on Theory and Practice in Information-Theoretic Security*, 2005.
- [53] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. In *Proc. of the Annual Technical Conf.*, 2004.
- [54] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. OpenDHT: A public DHT service and its uses. In *Proc. of ACM SIGCOMM*, 2005.
- [55] T. Ristenpart, G. Maganis, A. Krishnamurthy, and T. Kohno. Privacy-preserving location tracking of lost or stolen devices: Cryptographic techniques and replacing trusted third parties with DHTs. In *17th USENIX Security Symposium*, 2008.
- [56] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Lecture Notes in Computer Science*, 2001.
- [57] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proc. of Multimedia Computing and Networking*, 2002.
- [58] A. Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.
- [59] R. Singel. Encrypted e-mail company Hushmail spills to feds. <http://blog.wired.com/27bstroke6/2007/11/encrypted-e-mai.html>, 2007.
- [60] A. Singh, T. W. Ngan, P. Druschel, and D. S. Wallach. Eclipse attacks on overlay networks: Threats and defenses. In *Proc. of INFOCOM*, 2006.
- [61] Slashdot. <http://tech.slashdot.org/article.pl?sid=09/02/17/2213251&tid=267>, 2009.
- [62] Spitzer criminal complaint. <http://nytimes.com/packages/pdf/nyregion/20080310spitzer-complaint.pdf>, 2008.
- [63] M. Steiner and E. W. Biersack. Crawling Azureus. Technical Report RR-08-223, 2008.
- [64] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. of ACM SIGCOMM*, pages 149–160, 2001.
- [65] Y. Xie, F. Yu, K. Achan, E. Gillum, M. Goldszmidt, and T. Wobber. How dynamic are IP addresses? In *Proc. of SIGCOMM*, 2007.
- [66] K. Zetter. Tor researcher who exposed embassy e-mail passwords gets raided by Swedish FBI and CIA. <http://blog.wired.com/27bstroke6/2007/11/swedish-researc.html>, 2007.

Efficient Data Structures for Tamper-Evident Logging

Scott A. Crosby
scrosby@cs.rice.edu

Dan S. Wallach
dwallach@cs.rice.edu

Department of Computer Science, Rice University

Abstract

Many real-world applications wish to collect tamper-evident logs for forensic purposes. This paper considers the case of an untrusted logger, serving a number of clients who wish to store their events in the log, and kept honest by a number of auditors who will challenge the logger to prove its correct behavior. We propose semantics of tamper-evident logs in terms of this auditing process. The logger must be able to prove that individual logged events are still present, and that the log, as seen now, is consistent with how it was seen in the past. To accomplish this efficiently, we describe a tree-based data structure that can generate such proofs with logarithmic size and space, improving over previous linear constructions. Where a classic hash chain might require an 800 MB trace to prove that a randomly chosen event is in a log with 80 million events, our prototype returns a 3 KB proof with the same semantics. We also present a flexible mechanism for the log server to present authenticated and tamper-evident search results for all events matching a predicate. This can allow large-scale log servers to selectively delete old events, in an agreed-upon fashion, while generating efficient proofs that no inappropriate events were deleted. We describe a prototype implementation and measure its performance on an 80 million event syslog trace at 1,750 events per second using a single CPU core. Performance improves to 10,500 events per second if cryptographic signatures are offloaded, corresponding to 1.1 TB of logging throughput per week.

1 Introduction

There are over 10,000 U.S. regulations that govern the storage and management of data [22, 58]. Many countries have legal, financial, medical, educational and privacy regulations that require businesses to retain a variety of records. Logging systems are therefore in wide use (albeit many without much in the way of security features).

Audit logs are useful for a variety of forensic purposes, such as tracing database tampering [59] or building a versioned filesystem with verifiable audit trails [52]. Tamper-evident logs have also been used to build Byzantine fault-tolerant systems [35] and protocols [15], as well as to detect misbehaving hosts in distributed systems [28].

Ensuring a log's integrity is a critical component in the security of a larger system. Malicious users, including in-

siders with high-level access and the ability to subvert the logging system, may want to perform unlogged activities or tamper with the recorded history. While tamper-resistance for such a system might be impossible, tamper-detection should be guaranteed in a strong fashion.

A variety of hash data structures have been proposed in the literature for storing data in a tamper-evident fashion, such as trees [34, 49], RSA accumulators [5, 11], skip lists [24], or general authenticated DAGs. These structures have been used to build certificate revocation lists [49], to build tamper-evident graph and geometric searching [25], and authenticated responses to XML queries [19]. All of these store static data, created by a *trusted author* whose signature is used as a root-of-trust for authenticating responses of a lookup queries.

While authenticated data structures have been adapted for dynamic data [2], they continue to assume a trusted author, and thus they have no need to detect inconsistencies across versions. For instance, in SUNDR [36], a trusted network filesystem is implemented on untrusted storage. Although version vectors [16] are used to detect when the server presents forking-inconsistent views to clients, only trusted clients sign updates for the filesystem.

Tamper-evident logs are fundamentally different: An *untrusted* logger is the sole author of the log and is responsible for both building and signing it. A log is a dynamic data structure, with the author signing a stream of commitments, a new commitment each time a new event is added to the log. Each commitment *snaps* the entire log up to that point. If each signed commitment is the root of an authenticated data structure, well-known authenticated dictionary techniques [62, 42, 20] can detect tampering *within* each snapshot. However, without additional mechanisms to prevent it, an untrusted logger is free to have different snapshots make *inconsistent claims about the past*. To be secure, a tamper-evident log system must both detect tampering within each signed log *and* detect when different instances of the log make inconsistent claims.

Current solutions for detecting when an untrusted server is making inconsistent claims over time require linear space and time. For instance, to prevent undetected tampering, existing tamper evident logs [56, 17, 57] which rely upon a hash chain require auditors examine every intermediate event between snapshots. One proposal [43] for a tamper-evident log was based on a skip list. It has logarithmic lookup times, assuming the log

is known to be internally consistent. However, proving internal consistency requires scanning the full contents of the log. (See Section 3.4 for further analysis of this.)

In the same manner, CATS [63], a network-storage service with strong accountability properties, snapshots the internal state, and only probabilistically detects tampering by auditing a subset of objects for correctness between snapshots. Pavlou and Snodgrass [51] show how to integrate tamper-evidence into a relational database, and can prove the existence of tampering, if suspected. Auditing these systems for consistency is expensive, requiring each auditor visit each snapshot to confirm that any changes between snapshots are authorized.

If an untrusted logger knows that a just-added event or returned commitment will not be audited, then any tampering with the added event or the events fixed by that commitment will be undiscovered, and, by definition, the log is not tamper-evident. To prevent this, *a tamper-evident log requires frequent auditing*. To this end, we propose a tree-based history data structure, logarithmic for all auditing and lookup operations. Events may be added to the log, commitments generated, and audits may be performed independently of one another and at any time. No batching is used. Unlike past designs, we explicitly focus on how tampering will be discovered, through auditing, and we optimize the costs of these audits. Our *history tree* allows loggers to efficiently prove that the sequence of individual logs committed to, over time, make consistent claims about the past.

In Section 2 we present background material and propose semantics for tamper-evident logging. In Section 3 we present the history tree. In Section 4 we describe *Merkle aggregation*, a way to annotate events with attributes which can then be used to perform tamper-evident queries over the log and *safe deletion* of events, allowing unneeded events to be removed in-place, with no additional trusted party, while still being able to prove that no events were improperly purged. Section 5 describes a prototype implementation for tamper-evident logging of syslog data traces. Section 6 discusses approaches for scaling the logger's performance. Related work is presented in Section 7. Future work and conclusions appear in Section 8.

2 Security Model

In this paper, we make the usual cryptographic assumptions that an attacker cannot forge digital signatures or find collisions in cryptographic hash functions. Furthermore we are not concerned with protecting the secrecy of the logged events; this can be addressed with external techniques, most likely some form of encryption [50, 26, 54]. For simplicity, we assume a single monolithic log on a single host computer. Our goal is to detect tampering. It is impractical to prevent the destruction or alteration of

digital records that are in the custody of a Byzantine logger. Replication strategies, outside the scope of this paper, can help ensure availability of the digital records [44].

Tamper-evidence requires auditing. If the log is never examined, then tampering cannot be detected. To this end, we divide a logging system into three logical entities—many *clients* which generate events for appending to a log or history, managed on a centralized but totally untrusted *logger*, which is ultimately audited by one or more trusted *auditors*. We assume clients and auditors have very limited storage capacity while loggers are assumed to have unlimited storage. By auditing the published commitments and demanding proofs, auditors can be convinced that the log's integrity has been maintained. At least one auditor is assumed to be incorruptible. In our system, we distinguish between clients and auditors, while a single host could, in fact, perform both roles.

We must trust clients to behave correctly while they are following the event insertion protocol, but we trust clients nowhere else. Of course, a malicious client could insert garbage, but we wish to ensure that an event, once correctly inserted, cannot be undetectably hidden or modified, even if the original client is subsequently colluding with the logger in an attempt to tamper with old data.

To ensure these semantics, an untrusted logger must regularly prove its correct behavior to auditors and clients. *Incremental proofs*, demanded of the logger, prove that current commitment and prior commitment make consistent claims about past events. *Membership proofs* ask the logger to return a particular event from the log along with a proof that the event is consistent with the current commitment. Membership proofs may be demanded by clients after adding events or by auditors verifying that older events remain correctly stored by the logger. These two styles of proofs are sufficient to yield tamper-evidence. As any vanilla lookup operation may be followed by a request for proof, the logger must behave faithfully or risk its misbehavior being discovered.

2.1 Semantics of a tamper evident history

We now formalize our desired semantics for secure histories. Each time an event X is sent to the logger, it assigns an index i and appends it to the log, generating a version- i commitment C_i that depends on all of the events to-date, $X_0 \dots X_i$. The commitment C_i is bound to its version number i , signed, and published.

Although the stream of histories that a logger commits to ($C_0 \dots C_i, C_{i+1}, C_{i+2} \dots$) are supposed to be mutually-consistent, each commitment fixes an *independent* history. Because histories are not known, a priori, to be consistent with one other, we will use primes ($'$) to distinguish between different histories and the events contained within them. In other words, the events in log C_i (i.e., those committed by commitment C_i) are $X_0 \dots X_i$

and the events in $\log C'_j$ are $X'_0 \dots X'_j$, and we will need to prove their correspondence.

2.1.1 Membership auditing

Membership auditing is performed both by clients, verifying that new events are correctly inserted, and by auditors, investigating that old events are still present and unaltered. The logger is given an event index i and a commitment C_j , $i \leq j$ and is required to return the i th element in the log, X_i , and a proof that C_j implies X_i is the i th event in the log.

2.1.2 Incremental auditing

While a verified membership proof shows that an event was logged correctly in *some* log, represented by its commitment C_j , additional work is necessary to verify that the sequence of logs committed by the logger is consistent over time. In *incremental auditing*, the logger is given two commitments C_j and C'_k , where $j \leq k$, and is required to prove that the two commitments make consistent claims about past events. A verified incremental proof demonstrates that $X_a = X'_a$ for all $a \in [0, j]$. Once verified, the auditor knows that C_j and C'_k commit to the same shared history, and the auditor can safely discard C_j .

A dishonest logger may attempt to tamper with its history by rolling back the log, creating a new fork on which it inserts new events, and abandoning the old fork. Such tampering will be caught if the logging system satisfies *historical consistency* (see Section 2.3) and by a logger's inability to generate an incremental proof between commitments on different (and inconsistent) forks when challenged.

2.2 Client insertion protocol

Once clients receive commitments from the logger after inserting an event, they must immediately redistribute them to auditors. This prevents the clients from subsequently colluding with the logger to roll back or modify their events. To this end, we need a mechanism, such as a gossip protocol, to distribute the signed commitments from clients to multiple auditors. It's unnecessary for every auditor to audit every commitment, so long as some auditor audits every commitment. (We further discuss tradeoffs with other auditing strategies in Section 3.1.)

In addition, in order to deal with the logger presenting different views of the log to different auditors and clients, auditors must obtain and reconcile commitments received from multiple clients or auditors, perhaps with the gossip protocol mentioned above. Alternatively the logger may publish its commitment in a public fashion so that all auditors receive the same commitment [27]. All that matters is that auditors have access to a diverse collection of commitments and demand incremental proofs to verify that the logger is presenting a consistent view.

2.3 Definition: tamper evident history

We now define a tamper-evident history system as a five-tuple of algorithms:

$H.ADD(X) \rightarrow C_j$. Given an event X , appends it to the history, returning a new commitment.

$H.INCR.GEN(C_i, C_j) \rightarrow P$. Generates an incremental proof between C_i and C_j , where $i \leq j$.

$H.MEMBERSHIP.GEN(i, C_j) \rightarrow (P, X_i)$. Generates a membership proof for event i from commitment C_j , where $i \leq j$. Also returns the event, X_i .

$P.INCR.VF(C'_i, C_j) \rightarrow \{\top, \perp\}$. Checks that P proves that C_j fixes every entry fixed by C'_i (where $i \leq j$). Outputs \top if no divergence has been detected.

$P.MEMBERSHIP.VF(i, C_j, X'_i) \rightarrow \{\top, \perp\}$. Checks that P proves that event X'_i is the i 'th event in the log defined by C_j (where $i \leq j$). Outputs \top if true.

The first three algorithms run on the logger and are used to append to the log H and to generate *proofs* P . Auditors or clients verify the proofs with algorithms $\{INCR.VF, MEMBERSHIP.VF\}$. Ideally, the proof P sent to the auditor is more concise than retransmitting the full history H . Only commitments need to be signed by the logger. Proofs do not require digital signatures; either they demonstrate consistency of the commitments and the contents of an event or they don't. With these five operations, we now define "tamper evidence" as a system satisfying:

Historical Consistency If we have a valid incremental proof between two commitments C_j and C_k , where $j \leq k$, ($P.INCR.VF(C_j, C_k) \rightarrow \top$), and we have a valid membership proof P' for the event X'_i , where $i \leq j$, in the log fixed by C_j (i.e., $P'.MEMBERSHIP.VF(i, C_j, X'_i) \rightarrow \top$) and a valid membership proof for X''_i in the log fixed by C_k (i.e., $P''.MEMBERSHIP.VF(i, C_k, X''_i) \rightarrow \top$), then X'_i must equal X''_i . (In other words, if two commitments commit consistent histories, then they must both fix the same events for their shared past.)

2.4 Other threat models

Forward integrity Classic tamper-evident logging uses a different threat model, forward integrity [4]. The forward integrity threat model has two entities: clients who are fully trusted but have limited storage, and loggers who are assumed to be honest until suffering a Byzantine failure. In this threat model, the logger must be prevented from undetectably tampering with events logged prior to the Byzantine failure, but is allowed to undetectably tamper with events logged after the Byzantine failure.

Although we feel our threat model better characterizes the threats faced by tamper-evident logging, our history

tree and the semantics for tamper-evident logging are applicable to this alternative threat model with only minor changes. Under the semantics of forward-integrity, membership auditing just-added events is unnecessary because tamper-evidence only applies to events occurring before the Byzantine failure. Auditing a just-added event is unneeded if the Byzantine failure hasn't happened and irrelevant afterwards. Incremental auditing is still necessary. A client must incrementally audit received commitments to prevent a logger from tampering with events occurring before a Byzantine failure by rolling back the log and creating a new fork. Membership auditing is required to look up and examine old events in the log.

Itkis [31] has a similar threat model. His design exploited the fact that if a Byzantine logger attempts to roll back its history to before the Byzantine failure, the history must fork into two parallel histories. He proposed a procedure that tested two commitments to detect divergence without online interaction with the logger and proved an $O(n)$ lower bound on the commitment size. We achieve a tighter bound by virtue of the logger cooperating in the generation of these proofs.

Trusted hardware Rather than relying on auditing, an alternative model is to rely on the logger's hardware itself to be tamper-resistant [58, 1]. Naturally, the security of these systems rests on protecting the trusted hardware and the logging system against tampering by an attacker with complete physical access. Although our design could certainly use trusted hardware as an auditor, cryptographic schemes like ours rest on simpler assumptions, namely the logger can and must prove it is operating correctly.

3 History tree

We now present our new data structure for representing a tamper-evident history. We start with a Merkle tree [46], which has a long history of uses for authenticating static data. In a Merkle tree, data is stored at the leaves and the hash at the root is a tamper-evident summary of the contents. Merkle trees support logarithmic path lengths from the root to the leaves, permitting efficient random access. Although Merkle trees are a well-known tamper-evident data structure and our use is straightforward, the novelty in our design is in using a versioned computation of hashes over the Merkle tree to efficiently prove that different log snapshots, represented by Merkle trees, with *distinct* root hashes, make consistent claims about the past.

A filled history tree of depth d is a binary Merkle hash tree, storing 2^d events on the leaves. Interior nodes, $I_{i,r}$ are identified by their index i and layer r . Each leaf node $I_{i,0}$, at layer 0, stores event X_i . Interior node $I_{i,r}$ has left child $I_{i,r-1}$ and right child $I_{i+2^{r-1},r-1}$. (Figures 1 through 3 demonstrate this numbering scheme.) When a tree is not full, subtrees containing no events are

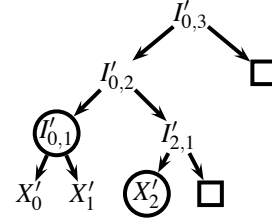


Figure 1: A version 2 history with commitment $C'_2 = I'_{0,3}$.

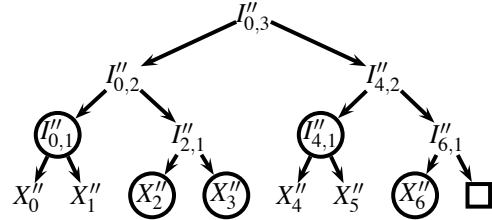


Figure 2: A version 6 history with commitment $C''_6 = I''_{0,3}$.

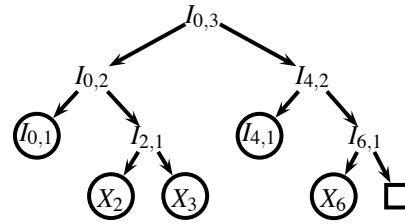


Figure 3: An incremental proof P between a version 2 and version 6 commitment. Hashes for the circled nodes are included in the proof. Other hashes can be derived from their children. Circled nodes in Figures 1 and 2 must be shown to be equal to the corresponding circled nodes here.

represented as \square . This can be seen starting in Figure 1, a version-2 tree having three events. Figure 2 shows a version-6 tree, adding four additional events. Although the trees in our figures have a depth of 3 and can store up to 8 leaves, our design clearly extends to trees with greater depth and more leaves.

Each node in the history tree is *labeled* with a cryptographic hash which, like a Merkle tree, fixes the contents of the subtree rooted at that node. For a leaf node, the label is the hash of the event; for an interior node, the label is the hash of the concatenation of the labels of its children.

An interesting property of the history tree is the ability to efficiently reconstruct old versions or *views* of the tree. Consider the history tree given in Figure 2. The logger could reconstruct C''_2 analogous to the version-2 tree in Figure 1 by pretending that nodes $I''_{4,2}$ and X''_3 were \square and then recomputing the hashes for the interior nodes and the root. If the reconstructed C''_2 matched a previously advertised commitment C'_2 , then both trees must have the same contents and commit the same events.

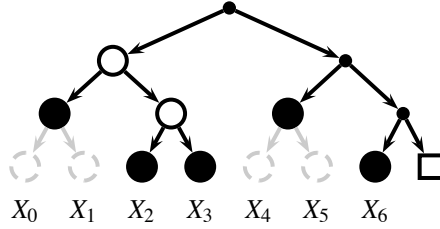


Figure 4: Graphical notation for a history tree analogous to the proof in Figure 3. Solid discs represent hashes included in the proof. Other nodes are not included. Dots and open circles represent values that can be recomputed from the values below them; dots may change as new events are added while open circles will not. Grey circle nodes are unnecessary for the proof.

This forms the intuition of how the logger generates an incremental proof P between two commitments, C'_2 and C''_6 . Initially, the auditor only possesses commitments C'_2 and C''_6 ; it does not know the underlying Merkle trees that these commitments fix. The logger must show that both histories commit the same events, i.e., $X''_0 = X'_0, X''_1 = X'_1$, and $X''_2 = X'_2$. To do this, the logger sends a *pruned tree* P to the auditor, shown in Figure 3. This pruned tree includes just enough of the full history tree to compute the commitments C'_2 and C''_6 . Unnecessary subtrees are *elided* out and replaced with *stubs*. Events can be either included in the tree or replaced by a stub containing their hash. Because an incremental proof involves *three* history trees, the trees committed by C'_2 and C''_6 with unknown contents and the pruned tree P , we distinguish them by using a different number of primes (').

From P , shown in Figure 3, we reconstruct the corresponding root commitment for a version-6 tree, C_6 . We recompute the hashes of interior nodes based on the hashes of their children until we compute the hash for node $I_{0,3}$, which will be the commitment C_6 . If $C''_6 = C_6$ then the corresponding nodes, circled in Figures 2 and 3, in the pruned tree P and the implicit tree committed by C''_6 must match.

Similarly, from P , shown in Figure 3, we can reconstruct the version-2 commitment C_2 by pretending that the nodes X_3 and $I_{4,2}$ are \square and, as before, recomputing the hashes for interior nodes up to the root. If $C'_2 = C_2$, then the corresponding nodes, circled in Figures 1 and 3, in the pruned tree P and the implicit tree committed by C'_2 must match, or $I''_{0,1} = I_{0,1}$ and $X''_2 = X_2$.

If the events committed by C'_2 and C''_6 are the same as the events committed by P , then they must be equal; we can then conclude that the tree committed by C''_6 is consistent with the tree committed by C'_2 . By this we mean that the history trees committed by C'_2 and C''_6 both commit the same events, or $X''_0 = X'_0, X''_1 = X'_1$, and $X''_2 = X'_2$, even though the events $X''_0 = X'_0, X''_1 = X'_1, X''_4$, and X''_5 are unknown to the auditor.

3.1 Is it safe to skip nodes during an audit?

In the pruned tree in Figure 3, we omit the events fixed by $I_{0,1}$, yet we still preserve the semantics of a tamper-evident log. Even though these earlier events may not be sent to the auditor, they are still fixed by the unchanged hashes above them in the tree. Any attempted tampering will be discovered in future incremental or membership audits of the skipped events. With the history tree, auditors only receive the portions of the history they need to audit the events they have chosen to audit. Skipping events makes it possible to conduct a variety of selective audits and offers more flexibility in designing auditing policies.

Existing tamper-evident log designs based on a classic hash-chain have the form $C_i = H(C_{i-1} \parallel X_i)$, $C_{-1} = \square$ and do not permit events to be skipped. With a hash chain, an incremental or membership proof between two commitments or between an event and a commitment must include *every* intermediate event in the log. In addition, because intermediate events cannot be skipped, each auditor, or client acting as an auditor, must eventually receive every event in the log. Hash chaining schemes, as such, are only feasible with low event volumes or in situations where every auditor is already receiving every event.

When membership proofs are used to investigate old events, the ability to skip nodes can lead to dramatic reductions in proof size. For example, in our prototype described in Section 5, in a log of 80 million events, our history tree can return a complete proof for any randomly chosen event in 3100 bytes. In a hash chain, where intermediate events cannot be skipped, an average of 40 million hashes would be sent.

Auditing strategies In many settings, it is possible that not every auditor will be interested in every logged event. Clients may not be interested in auditing events inserted or commitments received by other clients. One could easily imagine scenarios where a single logger is shared across many organizations, each only incentivized to audit the integrity of its own data. These organizations could run their own auditors, focusing their attention on commitments from their own clients, and only occasionally exchanging commitments with other organizations to ensure no forking has occurred. One can also imagine scenarios where independent accounting firms operate auditing systems that run against their corporate customers' log servers.

The log remains tamper-evident if clients gossip their received commitments from the logger to at least one honest auditor who uses it when demanding an incremental proof. By not requiring that every commitment be audited by every auditor, the total auditing overhead across all auditors can be proportional to the total number of events in the log—far cheaper than the number of events times the number of auditors as we might otherwise require.

$$A_{i,r}^v \equiv \text{FH}_{i,r} \quad \text{whenever } v \geq i + 2^r - 1 \quad (4)$$

$P.MEMBERSHIP.VF(i, C'_j, X'_i) \rightarrow \{\top, \perp\}$. From P apply equations (1)-(4) to compute $A_{0,d}^j$. Also extract X_i from the pruned tree P , which can only be done if P includes a path to event X_i . Return \top if $C'_j = A_{0,d}^j$ and $X_i = X'_i$.

Although incremental and membership proofs have different semantics, they both follow an identical tree structure and can be built and audited by a common implementation. In addition, a single pruned tree P can embed paths to several leaves to satisfy multiple auditing requests.

What is the size of a pruned tree used as a proof? The pruned tree necessary for satisfying a self-contained incremental proof between C_i and C_j or a membership proof for i in C_j requires that the pruned tree include a path to nodes X_i and X_j . This resulting pruned tree contains at most $2d$ frozen nodes, logarithmic in the size of the log.

In a real implementation, the log may have moved on to a later version, k . If the auditor requested an incremental proof between C_i and C_j , the logger would return the latest commitment C_k , and a pruned tree of at most $3d$ nodes, based around a version- k tree including paths to X_i and X_j . More typically, we expect auditors will request an incremental proof between a commitment C_i and the latest commitment. The logger can reply with the latest commitment C_k and pruned tree of at most $2d$ nodes that included a path to X_i .

The frozen hash cache In our description of the history tree, we described the *full representation* when we stated that the logger stores frozen hashes for all frozen interior nodes in the history tree. This cache is redundant whenever a node's hash can be recomputed from its children. We expect that logger implementations, which build pruned trees for audits and queries, will maintain and use the cache to improve efficiency.

When generating membership proofs, incremental proofs, and query lookup results, there is no need for the resulting pruned tree to include redundant hashes on interior nodes when they can be recomputed from their children. We assume that pruned trees used as proofs will use this *minimum representation*, containing frozen hashes only for stubs, to reduce communication costs.

Can overheads be reduced by exploiting redundancy between proofs? If an auditor is in regular communication with the logger, demanding incremental proofs between the previously seen commitment and the latest commitment, there is redundancy between the pruned subtrees on successive queries.

If an auditor previously requested an incremental proof between C_i and C_j and later requests an incremental proof P between C_j and C_n , the two proofs will share hashes on the path to leaf X_j . The logger may send a *partial proof* that omits these common hashes, and only contains the expected $O(\log_2(n - j))$ frozen hashes that are not shared

between the paths to X_j and X_n . This devolves to $O(1)$ if a proof is requested after every insertion. The auditor need only cache d frozen hashes to make this work.

Tree history time-stamping service Our history tree can be adapted to implement a round-based time-stamping service. After every round, the logger publishes the last commitment in public medium such as a newspaper. Let C_i be the commitment from the prior round and C_k be the commitment of the round a client requests that its document X_j be timestamped. A client can request a pruned tree including a path to leaves X_i, X_j, X_k . The pruned tree can be verified against the published commitments to prove that X_j was submitted in the round and its order within that round, without the cooperation of the logger.

If a separate history tree is built for each round, our history tree is equivalent to the threaded authentication tree proposed by Buldas et al. [10] for time-stamping systems.

3.3 Storing the log on secondary storage

Our history tree offers a curious property: it can be easily mapped onto write-once append-only storage. Once nodes become frozen, they become immutable, and are thus safe to output. This ordering is predetermined, starting with (X_0) , $(X_1, I_{0,1})$, (X_2) , $(X_3, I_{2,1}, I_{0,2})$, $(X_4) \dots$. Parentheses denote the nodes written by each ADD transaction. If nodes within each group are further ordered by their layer in the tree, this order is simply a post-order traversal of the binary tree. Data written in this linear fashion will minimize disk seek overhead, improving the disk's write performance. Given this layout, and assuming all events are the same size on disk, converting from an $(index, layer)$ to the byte index used to store that node takes $O(\log n)$ arithmetic operations, permitting efficient direct access.

In order to handle variable-length events, event data can be stored in a separate write-once append-only *value store*, while the leaves of the history tree contain offsets into the value store where the event contents may be found. Decoupling the history tree from the value store also allows many choices for how events are stored, such as databases, compressed files, or standard flat formats.

3.4 Comparing to other systems

In this section, we evaluate the time and space tradeoffs between our history tree and earlier hash chain and skip list structures. In all three designs, membership proofs have the same structure and size as incremental proofs, and proofs are generated in time proportional to their size.

Maniatis and Baker [43] present a tamper-evident log using a deterministic variant of a skip list [53]. The skip list history is like a hash-chain incorporating extra skip links that hop over many nodes, allowing for logarithmic lookups.

	Hash chain	Skip list	History tree
ADD Time	$O(1)$	$O(1)$	$O(\log_2 n)$
INCR.GEN proof size to C_k	$O(n - k)$	$O(n)$	$O(\log_2 n)$
MEMBERSHIP.GEN proof size for X_k	$O(n - k)$	$O(n)$	$O(\log_2 n)$
Cache size	-	$O(\log_2 n)$	$O(\log_2 n)$
INCR.GEN partial proof size	-	$O(n - j)$	$O(\log_2(n - j))$
MEMBERSHIP.GEN partial proof size	-	$O(\log_2(n - i))$	$O(\log_2(n - i))$

Table 1: We characterize the time to add an event to the log and the size of full and partial proofs generated in terms of n , the number of events in the log. For partial proofs audits, j denotes the number of events in the log at the time of the last audit and i denotes the index of the event being membership-audited.

In Table 1 we compare the three designs. All three designs have $O(1)$ storage per event and $O(1)$ commitment size. For skip list histories and tree histories, which support partial proofs (described in Section 3.2), we present the cache size and the expected proof sizes in terms of the number of events in the log, n , and the index, j , of the prior contact with the logger or the index i of the event being looked up. Our tree-based history strictly dominates both hash chains and skip lists in proof generation time and proof sizes, particularly when individual clients and auditors only audit a subset of the commitments or when partial proofs are used.

Canonical representation A hash chain history and our history tree have a canonical representation of both the history and of proofs within the history. In particular, from a given commitment C_n , there exists one unique path to each event X_i . When there are multiple paths auditing is more complex because the alternative paths must be checked for consistency with one another, both within a single history, and between the stream of histories C_i, C_{i+1}, \dots committed by the logger. Extra paths may improve the efficiency of looking up past events, such as in a skip list, or offer more functionality [17], but cannot be trusted by auditors and must be checked.

Maniatis and Baker [43] claim to support logarithmic-sized proofs, however they suffer from this multi-path problem. To verify internal consistency, an auditor with no prior contact with the logger must receive every event in the log in every incremental or membership proof.

Efficiency improves for auditors in regular contact with the logger that use partial proofs and cache $O(\log_2 n)$ state between incremental audits. If an auditor has previously verified the logger’s internal consistency up to C_j , the auditor will be able to verify the logger’s internal consistency up to a future commitment C_n with the receipt of events $X_{j+1} \dots X_n$. Once an auditor knows that the skip list is internally consistent the links that allow for logarithmic lookups can be trusted and subsequent membership proofs on old events will run in $O(\log_2 n)$ time. Skip list histories were designed to function in this mode, with each auditor eventually receiving every event in the log.

Auditing is required Hash chains and skip lists only offer a complexity advantage over the history tree when

adding new events, but this advantage is fleeting. If the logger knows that a given commitment will never be audited, it is free to tamper with the events fixed by that commitment, and the log is no longer provably tamper evident. Every commitment returned by the logger must have a non-zero chance of being audited and any evaluation of tamper-evident logging must include the costs of this unavoidable auditing. With multiple auditors, auditing overhead is further multiplied. After inserting an event, hash chains and skip lists suffer an $O(n - j)$ disadvantage the moment they do incremental audits between the returned commitment and prior commitments. They cannot reduce this overhead by, for example, only auditing a random subset of commitments.

Even if the threat model is weakened from our always-untrusted logger to the forward-integrity threat model (See Section 2.4), hash chains and skip lists are less efficient than the history tree. Clients can forgo auditing just-added events, but are still required to do incremental audits to prior commitments, which are expensive with hash chains or skip lists.

4 Merkle aggregation

Our history tree permits $O(\log_2 n)$ access to arbitrary events, given their index. In this section, we extend our history tree to support efficient, tamper-evident content searches through a feature we call *Merkle aggregation*, which encodes auxiliary information into the history tree. Merkle aggregation permits the logger to perform authorized purges of the log while detecting unauthorized deletions, a feature we call *safe deletion*.

As an example, imagine that a client flags certain events in the log as “important” when it stores them. In the history tree, the logger propagates these flags to interior nodes, setting the flag whenever either child is flagged. To ensure that the tagged history is tamper-evident, this flag can be incorporated into the hash label of a node and checked during auditing. As clients are assumed to be trusted when inserting into the log, we assume clients will properly annotate their events. Membership auditing will detect if the logger incorrectly stored a leaf with the wrong flag or improperly propagated the flag. Incremental audits would detect tampering if any frozen

node had its flag altered. Now, when an auditor requests a list of only flagged events, the logger can generate that list along with a proof that the list is complete. If there are relatively few “important” events, the query results can skip over large chunks of the history.

To generate a proof that the list of flagged events is complete, the logger traverses the full history tree H , pruning any subtrees without the flag set, and returns a pruned tree P containing only the visited nodes. The auditor can ensure that no flagged nodes were omitted in P by performing its own recursive traversal on P and verifying that every stub is unflagged.

Figure 7 shows the pruned tree for a query against a version-5 history with events X_2 and X_5 flagged. Interior nodes in the path from X_2 and X_5 to the root will also be flagged. For subtrees containing no matching events, such as the parent of X_0 and X_1 , we only need to retain the root of the subtree to vouch that its children are unflagged.

4.1 General attributes

Boolean flags are only one way we may flag log events for later queries. Rather than enumerate every possible variation, we abstract an aggregation strategy over attributes into a 3-tuple, (τ, \oplus, Γ) . τ represents the type of attribute or attributes that an event has. \oplus is a deterministic function used to compute the attributes on an interior node in the history tree by *aggregating* the attributes of the node’s children. Γ is a deterministic function that maps an event to its attributes. In our example of client-flagged events, the aggregation strategy is $(\tau := \text{BOOL}, \oplus := \vee, \Gamma(x) := x.\text{isFlagged})$.

For example, in a banking application, an attribute could be the dollar value of a transaction, aggregated with the MAX function, permitting queries to find all transactions over a particular dollar value and detect if the logger tampers with the results. This corresponds to $(\tau := \text{INT}, \oplus := \text{MAX}, \Gamma(x) := x.\text{value})$. Or, consider events having internal timestamps, generated by the client, arriving at the logger out of order. If we attribute each node in the tree with the earliest and latest timestamp found among its children, we can now query the logger for all nodes within a given time range, regardless of the order of event arrival.

There are at least three different ways to implement keyword searching across logs using Merkle aggregation. If the number of keywords is fixed in advance, then the attribute τ for events can be a bit-vector or sparse bit-vector combined with $\oplus := \vee$. If the number of keywords is unknown, but likely to be small, τ can be a sorted list of keywords, with $\oplus := \cup$ (set union). If the number of keywords is unknown and potentially unbounded, then a Bloom filter [8] may be used to represent them, with τ being a bit-vector and $\oplus := \vee$. Of course, the Bloom filter would then have the potential of returning false positives to a query, but there would be no false negatives.

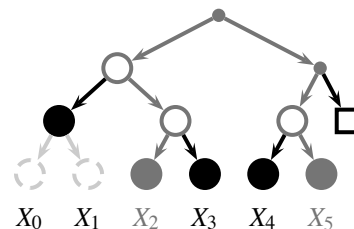


Figure 7: Demonstration of Merkle aggregation with some events flagged as important (highlighted). Frozen nodes that would be included in a query are represented as solid discs.

Merkle aggregation is extremely flexible because Γ can be *any* deterministic computable function. However, once a log has been created, (τ, \oplus, Γ) are fixed for that log, and the set of queries that can be made is restricted based on the aggregation strategy chosen. In Section 5 we describe how we were able to apply these concepts to the metadata used in Syslog logs.

4.2 Formal description

To make attributes tamper-evident in history trees, we modify the computation of hashes over the tree to include them. Each node now has a hash label denoted by $A_{i,r}^v.H$ and an annotation denoted by $A_{i,r}^v.A$ for storing attributes. Together these form the node data that is attached to each node in the history tree. Note that the hash label of node, $A_{i,r}^v.H$, does *not* fix its own attributes, $A_{i,r}^v.A$. Instead, we define a *subtree authenticator* $A_{i,r}^v.* = H(A_{i,r}^v.H \parallel A_{i,r}^v.A)$ that fixes the attributes and hash of a node, and recursively fixes every hash and attribute in its subtree. Frozen hashes $FH_{i,r}.A$ and $FH_{i,r}.H$ and $FH_{i,r}.*$ are defined analogously to the non-Merkle-aggregation case.

We could have defined this recursion in several different ways. This representation allows us to elide unwanted subtrees with a small stub, containing one hash and one set of attributes, while exposing the attributes in a way that makes it possible to locally detect if the attributes were improperly aggregated.

Our new mechanism for computing hash and aggregates for a node is given in equations (5)-(10) in Figure 8. There is a strong correspondence between this recurrence and the previous one in Figure 5. Equations (6) and (7) extract the hash and attributes of an event, analogous to equation (1). Equation (9) handles aggregation of attributes between a node and its children. Equation (8) computes the hash of a node in terms of the subtree authenticators of its children.

INCR.GEN and MEMBERSHIP.GEN operate the same as with an ordinary history tree, except that wherever a frozen hash was included in the proof ($FH_{i,r}$), we now include both the hash of the node, $FH_{i,r}.H$, and its attributes $FH_{i,r}.A$. Both are required for recomputing $A_{i,r}^v.A$ and $A_{i,r}^v.H$ for the parent node. ADD, INCR.VF,

$$A_{i,r}^v.* = H(A_{i,r}^v.H \| A_{i,r}^v.A) \quad (5)$$

$$A_{i,0}^v.H = \begin{cases} H(0 \| X_i) & \text{if } v \geq i \end{cases} \quad (6)$$

$$A_{i,0}^v.A = \begin{cases} \Gamma(X_i) & \text{if } v \geq i \end{cases} \quad (7)$$

$$A_{i,r}^v.H = \begin{cases} H(1 \| A_{i,r-1}^v.* \| \square) & \text{if } v < i + 2^{r-1} \\ H(1 \| A_{i,r-1}^v.* \| A_{i+2^{r-1},r-1}^v.*) & \text{if } v \geq i + 2^{r-1} \end{cases} \quad (8)$$

$$A_{i,r}^v.A = \begin{cases} A_{i,r-1}^v.A & \text{if } v < i + 2^{r-1} \\ A_{i,r-1}^v.A \oplus A_{i+2^{r-1},r-1}^v.A & \text{if } v \geq i + 2^{r-1} \end{cases} \quad (9)$$

$$C_n = A_{0,d}^n.* \quad (10)$$

Figure 8: Hash computations for Merkle aggregation

and `MEMBERSHIP.VF` are the same as before except for using the equations (5)-(10) for computing hashes and propagating attributes. Merkle aggregation inflates the storage and proof sizes by a factor of $(A + B)/A$ where A is the size of a hash and B is the size of the attributes.

4.2.1 Queries over attributes

In Merkle aggregation queries, we permit query results to contain false positives, i.e., events that do not match the query Q . Extra false positive events in the result only impact performance, not correctness, as they may be filtered by the auditor. We forbid false negatives; every event matching Q will be included in the result.

Unfortunately, Merkle aggregation queries can only match attributes, not events. Consequently, we must conservatively transform a query Q over events into a predicate Q^Γ over attributes and require that it be *stable*, with the following properties: If Q matches an event then Q^Γ matches the attributes of that event (i.e., $\forall_x Q(x) \Rightarrow Q^\Gamma(\Gamma(x))$). Furthermore, if Q^Γ is true for either child of a node, it must be true for the node itself (i.e., $\forall_{x,y} Q^\Gamma(x) \vee Q^\Gamma(y) \Rightarrow Q^\Gamma(x \oplus y)$ and $\forall_x Q^\Gamma(x) \vee Q^\Gamma(\square) \Rightarrow Q^\Gamma(x \oplus \square)$).

Stable predicates can falsely match nodes or events for two reasons: events' attributes may match Q^Γ without the events matching Q , or nodes may occur where $(Q^\Gamma(x) \vee Q^\Gamma(y))$ is false, but $Q^\Gamma(x \oplus y)$ is true. We call a predicate Q *exact* if there can be no false matches. This occurs when $Q(x) \Leftrightarrow Q^\Gamma(\Gamma(x))$ and $Q^\Gamma(x) \vee Q^\Gamma(y) \Leftrightarrow Q^\Gamma(x \oplus y)$. Exact queries are more efficient because a query result does not include falsely matching events and the corresponding pruned tree proving the correctness of the query result does not require extra nodes.

Given these properties, we can now define the additional operations for performing authenticated queries on the log for events matching a predicate Q^Γ .

$H.QUERY(C_j, Q^\Gamma) \rightarrow P$ Given a predicate Q^Γ over attributes τ , returns a pruned tree where every elided

subtrees does not match Q^Γ .

$P.QUERY.VF(C_j', Q^\Gamma) \rightarrow \{\top, \perp\}$ Checks the pruned tree P and returns \top if every stub in P does not match Q^Γ and the reconstructed commitment C_j is the same as C_j' .

Building a pruned tree containing all events matching a predicate Q^Γ is similar to building the pruned trees for membership or incremental auditing. The logger starts with a proof skeleton then recursively traverses it, splitting interior nodes when $Q^\Gamma(FH_{i,r}.A)$ is true. Because the predicate Q^Γ is stable, no event in any elided subtree can match the predicate. If there are t events matching the predicate Q^Γ , the pruned tree is of size at most $O((1+t)\log_2 n)$ (i.e., t leaves with $\log_2 n$ interior tree nodes on the paths to the root).

To verify that P includes all events matching Q^Γ , the auditor does a recursive traversal over P . If the auditor finds an interior stub where $Q^\Gamma(FH_{i,r}.A)$ is true, the verification fails because the auditor found a node that was supposed to have been split. (Unfrozen nodes will always be split as they compose the proof skeleton and only occur on the path from X_j to the root.) The auditor must also verify that pruned tree P commits the same events as the commitment C_j' by reconstructing the root commitment C_j using the equations (5)-(10) and checking that $C_j = C_j'$.

As with an ordinary history tree, a Merkle aggregating tree requires auditing for tamper-detection. If an event is never audited, then there is no guarantee that its attributes have been properly included. Also, a dishonest logger or client could deliberately insert false log entries whose attributes are aggregated up the tree to the root, causing garbage results to be included in queries. Even so, if Q is stable, a malicious logger cannot hide matching events from query results without detection.

4.3 Applications

Safe deletion Merkle aggregation can be used for expiring old and obsolete events that do not satisfy some predicate and prove that no other events were deleted inappropriately. While Merkle aggregation queries prove that no matching event is excluded from a query result, safe deletion requires the contrapositive: proving to an auditor that each purged event was legitimately purged because it did not match the predicate.

Let $Q(x)$ be a stable query that is true for all events that the logger must keep. Let $Q^\Gamma(x)$ be the corresponding predicate over attributes. The logger stores a pruned tree that includes all nodes and leaf events where $Q^\Gamma(x)$ is true. The remaining nodes may be elided and replaced with stubs. When a logger cannot generate a path to a previously deleted event X_i , it instead supplies a pruned tree that includes a path to an ancestor node A of X_i where $Q^\Gamma(A)$ is false. Because Q is stable, if $Q^\Gamma(A)$ is false, then $Q^\Gamma(\Gamma(X_i))$ and $Q(X_i)$ must also be false.

Safe deletion and auditing policies must take into account that if a subtree containing events $X_i \dots X_j$ is purged, the logger is unable to generate incremental or membership proofs involving commitments $C_i \dots C_j$. The auditing policy must require that any audits using those commitments be performed before the corresponding events are deleted, which may be as simple as requiring that clients periodically request an incremental proof to a later or long-lived commitment.

Safe deletion will not save space when using the append-only storage described in Section 3.3. However, if data-destruction policies require destroying a subset of events in the log, safe deletion may be used to prove that no unauthorized log events were destroyed.

“Private” search Merkle aggregation enables a weak variant of private information retrieval [14], permitting clients to have privacy for the specific contents of their events. To aggregate the attributes of an event, the logger only needs the attributes of an event, $\Gamma(X_i)$, not the event itself. To verify that aggregation is done correctly also only requires the attributes of an event. If clients encrypt their events and digitally sign their public attributes, auditors may verify that aggregation is done correctly while clients preserve their event privacy from the logger and other clients and auditors.

Bloom filters, in addition to providing a compact and approximate way to represent the presence or absence of a large number of keywords, can also enable private indexing (see, e.g., Goh [23]). The logger has no idea what the individual keywords are within the Bloom filter; many keywords could map to the same bit. This allows for private keywords that are still protected by the integrity mechanisms of the tree.

5 Syslog prototype implementation

Syslog is the standard Unix-based logging system [38], storing events with many attributes. To demonstrate the effectiveness of our history tree, we built an implementation capable of storing and searching syslog events. Using events from syslog traces, captured from our departmental servers, we evaluated the storage and performance costs of tamper-evident logging and secure deletion.

Each syslog event includes a timestamp, the host generating the event, one of 24 *facilities* or subsystem that generated the event, one of 8 logging *levels*, and the *message*. Most events also include a *tag* indicating the program generating the event. Solutions for authentication, management, and reliable delivery of syslog events over the network have already been proposed [48] and are in the process of being standardized [32], but none of this work addresses the logging semantics that we wish to provide.

Our prototype implementation was written in a hybrid of Python 2.5.2 and C++ and was benchmarked on an

Intel Core 2 Duo 2.4GHz CPU with 4GB of RAM in 64-bit mode under Linux. Our present implementation is single-threaded, so the second CPU core is underutilized. Our implementation uses SHA-1 hashes and 1024-bit DSA signatures, borrowed from the OpenSSL library.

In our implementation, we use the array-based post-order traversal representation discussed in Section 3.3. The value store and history tree are stored in separate write-once append-only files and mapped into memory. Nodes in the history tree use a fixed number of bytes, permitting direct access. Generating membership and incremental proofs requires RAM proportional to the size of the proof, which is logarithmic in the number of events in the log. Merkle aggregation query result sizes are presently limited to those which can fit in RAM, approximately 4 million events.

The storage overheads of our tamper-evident history tree are modest. Our prototype stores five attributes for each event. Tags and host names are encoded as 2-of-32 bit Bloom filters. Facilities and hosts are encoded as bit-vectors. To permit range queries to find every event in a particular range of time, an interval is used to encode the message timestamp. All together, there are twenty bytes of attributes and twenty bytes for a SHA-1 hash for each node in the history tree. Leaves have an additional twelve bytes to store the offset and length of the event contents in the value store.

We ran a number of simulations of our prototype to determine the processing time and space overheads of the history tree. To this end, we collected a trace of four million events from thirteen of our departmental server hosts over 106 hours. We observed 9 facilities, 6 levels, and 52 distinct tags. 88.1% of the events are from the mail server and 11.5% are from 98,743 failed ssh connection attempts. Only .393% of the log lines are from other sources. In testing our history tree, we replay this trace 20 times to insert 80 million events. Our syslog trace, after the replay, occupies 14.0 GB, while the history tree adds an additional 13.6 GB.

5.1 Performance of the logger

The logger is the only centralized host in our design and may be a bottleneck. The performance of a real world logger will depend on the auditing policy and relative frequency between inserting events and requesting audits. Rather than summarize the performance of the logger for one particular auditing policy, we benchmark the costs of the various tasks performed by the logger.

Our captured syslog traces averaged only ten events per second. Our prototype can insert events at a rate of 1,750 events per second, including DSA signature generation. Inserting an event requires four steps, shown in Table 2, with the final step, signing the resulting commitment, responsible for most of the processing time. Throughput

Step	Task	% of CPU	Rate (events/sec)
A	Parse syslog message	2.4%	81,000
B	Insert event into log	2.6%	66,000
C	Generate commitment	11.8%	15,000
D	Sign commitment	83.3%	2,100
	Membership proofs (with locality)	-	8,600
	Membership proofs (no locality)	-	32

Table 2: Performance of the logger in each of the four steps required to insert an event and sign the resulting commitment and in generating membership proofs. Rates are given assuming nothing other than the specified step is being performed.

would increase to 10,500 events per second if the DSA signatures were computed elsewhere (e.g., leveraging multiple CPU cores). (Section 6 discusses scalability in more detail.) This corresponds to 1.9MB/sec of uncompressed syslog data (1.1 TB per week).

We also measured the rate at which our prototype can generate membership and incremental proofs. The size of an incremental proof between two commitments depends upon the distance between the two commitments. As the distance varies from around two to two million events, the size of a self-contained proof varies from 1200 bytes to 2500 bytes. The speed for generating these proofs varies from 10,500 proofs/sec to 18,000 proofs/sec, with shorter distances having smaller proof sizes and faster performance than longer distances. For both incremental and membership proofs, compressing by gzip [18] halves the size of the proofs, but also halves the rate at which proofs can be generated.

After inserting 80 million events into the history tree, the history tree and value store require 27 GB, several times larger than our test machine’s RAM capacity. Table 2 presents our results for two membership auditing scenarios. In our first scenario we requested membership proofs for random events chosen among the most recent 5 million events inserted. Our prototype generated 8,600 self-contained membership proofs per second, averaging 2,400 bytes each. In this high-locality scenario, the most recent 5 million events were already sitting in RAM. Our second scenario examined the situation when audit requests had low locality by requesting membership proofs for random events anywhere in the log. The logger’s performance was limited to our disk’s seek latency. Proof size averaged 3,100 bytes and performance degraded to 32 membership proofs per second. (We discuss how this might be overcome in Section 6.2.)

To test the scalability of the history tree, we benchmarked insert performance and auditing performance on our original 4 million event syslog event trace, without replication, and the 80 million event trace after 20x replication. Event insertion and incremental auditing are

roughly 10% slower on the larger log.

5.2 Performance of auditors and clients

The history tree places few demands upon auditors or clients. Auditors and clients must verify the logger’s commitment signatures and must verify the correctness of pruned tree replies to auditing requests. Our machine can verify 1,900 DSA-1024 signatures per second. Our current tree parser is written in Python and is rather slow. It can only parse 480 pruned trees per second. Once the pruned tree has been parsed, our machine can verify 9,000 incremental or membership proofs per second. Presently, one auditor cannot verify proofs as fast as the logger can generate them, but auditors can clearly operate independently of one another, in parallel, allowing for exceptional scaling, if desired.

5.3 Merkle aggregation results

In this subsection, we describe the benefits of Merkle aggregation in generating query results and in safe deletion. In our experiments, due to limitations of our implementation in generating large pruned trees, our Merkle aggregation experiments used the smaller four million event log.

We used 86 different predicates to investigate the benefits of safe deletion and the overheads of Merkle aggregation queries. We used 52 predicates, each matching one tag, 13 predicates, each matching one host, 9 predicates, each matching one facility, 6 predicates, one matching each level, and 6 predicates, each matching the k highest logging levels.

The predicates matching tags and hosts use Bloom filters, are *inexact*, and may have false positives. This causes 34 of the 65 Bloom filter query results to include more nodes than our “worst case” expectation for exact predicates. By using larger Bloom filters, we reduce the chances of spurious matches. When a 4-of-64 Bloom filter is used for tags and hostnames, pruned trees resulting from search queries average 15% fewer nodes, at the cost of an extra 64 bits of attributes for each node in the history tree. In a real implementation, the exact parameters of the Bloom filter would best be tuned to match a sample of the events being logged.

Merkle aggregation and safe deletion Safe deletion allows the purging of unwanted events from the log. Auditors define a stable predicate over the attributes of events indicating which events must be kept, and the logger keeps a pruned tree of only those matching events. In our first test, we simulated the deletion of all events except those from a particular host. The pruned tree was generated in 14 seconds, containing 1.92% of the events in the full log and serialized to 2.29% of the size of the full tree. Although 98.08% of the events were purged, the logger was only able to purge 95.1% of the nodes in the

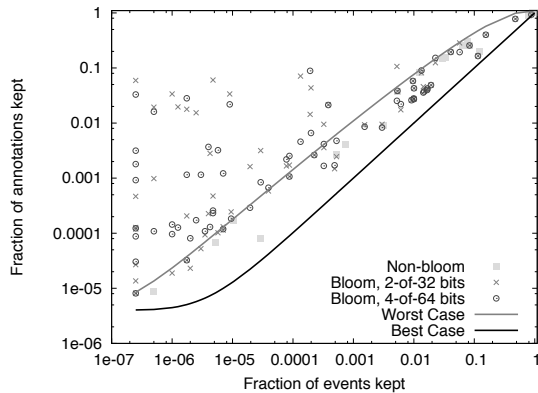


Figure 9: Safe deletion overhead. For a variety of queries, we plot the fraction of hashes and attributes kept after deletion versus the fraction of events kept.

history tree because the logger must keep the hash label and attributes for the root nodes of elided subtrees.

When measuring the size of a pruned history tree generated by safe deletion, we assume the logger caches hashes and attributes for all interior nodes in order to be able to quickly generate proofs. For each predicate, we measure the *kept ratio*, the number of interior node or stubs in a pruned tree of all nodes matching the predicate divided by the number of interior nodes in the full history tree. In Figure 9 for each predicate we plot the kept ratio versus the fraction of events matching the predicate. We also plot the analytic best-case and worst-case bounds, based on a continuous approximation. The minimum overhead occurs when the matching events are contiguous in the log. The worst-case occurs when events are maximally separated in the log. Our Bloom-filter queries do worse than the “worst-case” bound because Bloom filter matches are inexact and will thus trigger false positive matches on interior nodes, forcing them to be kept in the resulting pruned tree. Although many Bloom filters did far worse than the “worst-case,” among the Bloom filters that matched fewer than 1% of the events in the log, the logger is still able to purge over 90% of the nodes in the history tree and often did much better than that.

Merkle aggregation and authenticated query results

In our second test, we examine the overheads for Merkle aggregation query lookup results. When the logger generates the results to a query, the resulting pruned tree will contain both matching events and history tree overhead, in the form of hashes and attributes for any stubs. For each predicate, we measure the *query overhead ratio*—the number of stubs and interior nodes in a pruned tree divided by the number of events in the pruned tree. In Figure 10 we plot the query overhead ratio versus the fraction of events matching the query for each of our 86 predicates. This plot shows, for each event matching a predicate, proportionally how much extra overhead is in-

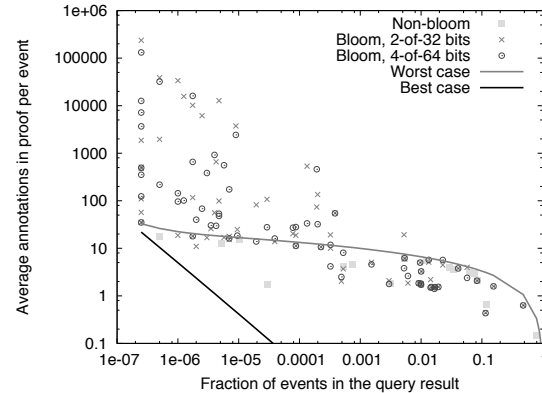


Figure 10: Query overhead per event. We plot the ratio between the number of hashes and matching events in the result of each query versus the fraction of events matching the query.

curred, per event, for authentication information. We also plot the analytic best-case and worst-case bounds, based on a continuous approximation. The minimum overhead occurs when the matching events are contiguous in the log. The worst-case occurs when events are maximally separated in the log. With exact predicates, the overhead of authenticated query results is very modest, and again, inexact Bloom filter queries will sometimes do worse than the “worst case.”

6 Scaling a tamper-evident log

In this section, we discuss techniques to improve the insert throughput of the history tree by using concurrency, and to improve the auditing throughput with replication. We also discuss a technique to amortize the overhead of a digital signature over several events.

6.1 Faster inserts via concurrency

Our tamper-evident log offers many opportunities to leverage concurrency to increase throughput. Perhaps the simplest approach is to offload signature generation. From Table 2, signatures account for over 80% of the runtime cost of an insert. Signatures are not included in any other hashes and there are no interdependencies between signature computations. Furthermore, signing a commitment does not require knowing anything other than the root commitment of the history tree. Consequently, it’s easy to offload signature computations onto additional CPU cores, additional hosts, or hardware crypto accelerators to improve throughput.

It is possible for a logger to also generate commitments concurrently. If we examine Table 2, parsing and inserting events in the log is about two times faster than generating commitments. Like signatures, commitments have no interdependencies on one other; they depend only on the history tree contents. As soon as event X_j is inserted into the tree and $O(1)$ frozen hashes are computed and stored,

a new event may be immediately logged. Computing the commitment C_j only requires read-only access to the history tree, allowing it to be computed concurrently by another CPU core without interfering with subsequent events. By using shared memory and taking advantage of the append-only write-once semantics of the history tree, we would expect concurrency overhead to be low.

We have experimentally verified the maximum rate at which our prototype implementation, described in Section 5, can insert syslog events into the log at 38,000 events per second using only one CPU core on commodity hardware. This is the maximum throughput our hardware could potentially support. In this mode we assume that digital signatures, commitment generation, and audit requests are delegated to additional CPU cores or hosts. With multiple hosts, each host must build a replica of the history tree which can be done at least as fast as our maximum insert rate of 38,000 events per second. Additional CPU cores on these hosts can then be used for generating commitments or handling audit requests.

For some applications, 38,000 events per second may still not be fast enough. Scaling beyond this would require fragmenting the event insertion and storage tasks across multiple logs. To break interdependencies between them, the fundamental history tree data structure we presently use would need to evolve, perhaps into disjoint logs that occasionally entangle with one another as in timeline entanglement [43]. Designing and evaluating such a structure is future work.

6.2 Logs larger than RAM

For exceptionally large audits or queries, where the working set size does not fit into RAM, we observed that throughput was limited to disk seek latency. Similar issues occur in any database query system that uses secondary storage, and the same software and hardware techniques used by databases to speed up queries may be used, including faster or higher throughput storage systems or partitioning the data and storing it in-memory across a cluster of machines. A single large query can then be issued to the cluster node managing each sub-tree. The results would then be merged before transmitting the results to the auditor. Because each sub-tree would fit in its host's RAM, sub-queries would run quickly.

6.3 Signing batches of events

When large computer clusters are unavailable and the performance cost of DSA signatures is the limiting factor in the logger's throughput, we may improve performance of the logger by allowing multiple updates to be handled with one signature computation.

Normally, when a client requests an event X to be inserted, the logger assigns it an index i , generates the commitment C_i , signs it, and returns the result. If the

logger has insufficient CPU to sign every commitment, the logger could instead delay returning C_i until it has a signature for some later commitment C_j ($j \geq i$). This later signed commitment could then be sent to the client expecting an earlier one. To ensure that the event X_i in the log committed by C_j was X , the client may request a membership proof from commitment C_j to event i and verify that $X_i = X$. This is safe due to the tamper-evidence of our structure. If the logger were ever to later sign a C_i inconsistent with C_j , it would fail an incremental proof.

In our prototype, inserting events into the log is twenty times faster than generating and signing commitments. The logger may amortize the costs of generating a signed commitment over many inserted events. The number of events per signed commitment could vary dynamically with the load on the logger. Under light load, the logger could sign every commitment and insert 1,750 events per second. With increasing load, the logger might sign one in every 16 commitments to obtain an estimated insert rate of 17,000 events per second. Clients will still receive signed commitments within a fraction of a second, but several clients can now receive the same commitment. Note that this analysis only considers the maximum insert rate for the log and does not include the costs of replying to audits. The overall performance improvements depend on how often clients request incremental and membership proofs.

7 Related work

There has been recent interest in creating append-only databases for regulatory compliance. These databases permit the ability to access old versions and trace tampering [51]. A variety of different data structures are used, including a B-tree [64] and a full text index [47]. The security of these systems depends on a write-once semantics of the underlying storage that cannot be independently verified by a remote auditor.

Forward-secure digital signature schemes [3] or stream authentication [21] can be used for signing commitments in our scheme or any other logging scheme. Entries in the log may be encrypted by clients for privacy. Kelsey and Schneier [57] have the logger encrypt entries with a key destroyed after use, preventing an attacker from reading past log entries. A hash function is iterated to generate the encryption keys. The initial hash is sent to a trusted auditor so that it may decrypt events. Logcrypt [29] extends this to public key cryptography.

Ma and Tsudik [41] consider tamper-evident logs built using forward-secure sequential aggregating signature schemes [39, 40]. Their design is round-based. Within each round, the logger evolves its signature, combining a new event with the existing signature to generate a new signature, and also evolves the authentication key. At the end of a round, the final signature can authenticate any event inserted.

Davis et. al. [17] permits keyword searching in a log by trusting the logger to build parallel hash chains for each keyword. Techniques have also been designed for keyword searching encrypted logs [60, 61]. A tamper-evident store for voting machines has been proposed, based on append-only signatures [33], but the signature sizes grow with the number of signed messages [6].

Many timestamping services have been proposed in the literature. Haber and Stornetta [27] introduce a timestamping service based on hash chains, which influenced the design of Surety, a commercial timestamping service that publishes their head commitment in a newspaper once a week. Chronos is a digital timestamping service inspired by a skip list, but with a hashing structure similar to our history tree [7]. This and other timestamping designs [9, 10] are round-based. In each round, the logger collects a set of events and stores the events within that round in a tree, skip list, or DAG. At the end of the round the logger publicly broadcasts (e.g., in a newspaper) the commitment for that round. Clients then obtain a logarithmically-sized, tamper-evident proof that their events are stored within that round and are consistent with the published commitment. Efficient algorithms have been constructed for outputting time stamp authentication information for successive events within a round in a streaming fashion, with minimal storage on the server [37]. Unlike these systems, our history tree allows events to be added to the log, commitments generated, and audits to be performed at any time.

Maniatis and Baker [43] introduced the idea of *timeline entanglement*, where every participant in a distributed system maintains a log. Every time a message is received, it is added to the log, and every message transmitted contains the hash of the log head. This process spreads commitments throughout the network, making it harder for malicious nodes to diverge from the canonical timeline without there being evidence somewhere that could be used in an audit to detect tampering. Auditorium [55] uses this property to create a shared “bulletin board” that can detect tampering even when $N - 1$ systems are faulty.

Secure aggregation has been investigated as a distributed protocol in sensor networks for computing sums, medians, and other aggregate values when the host doing the aggregation is not trusted. Techniques include trading off approximate results in return for sublinear communication complexity [12], or using MAC codes to detect one-hop errors in computing aggregates [30]. Other aggregation protocols have been based around hash tree structures similar to the ones we developed for Merkle aggregation. These structures combine aggregation and cryptographic hashing, and include distributed sensor-network aggregation protocols for computing authenticated sums [13] and generic aggregation [45]. The sensor network aggregation protocols interactively gener-

ate a secure aggregate of a set of measurements. In Merkle aggregation, we use intermediate aggregates as a tool for performing efficient queries. Also, our Merkle aggregation construction is more efficient than these designs, requiring fewer cryptographic hashes to verify an event.

8 Conclusions

In this work we have shown that regular and continuous auditing is a critical operation for any tamper-evident log system, for without auditing, clients cannot detect if a Byzantine logger is misbehaving by not logging events, removing unaudited events, or forking the log. From this requirement we have developed a new tamper-evident log design, based on a new Merkle tree data structure that permits a logger to produce concise proofs of its correct behavior. Our system eliminates any need to trust the logger, instead allowing clients and auditors of the logger to efficiently verify its correct behavior with only a constant amount of local state. By sharing commitments among clients and auditors, our design is resistant even to sophisticated forking or rollback attacks, even in cases where a client might change its mind and try to repudiate events that it had logged earlier.

We also proposed Merkle aggregation, a flexible mechanism for encoding auxiliary attributes into a Merkle tree that allows these attributes to be aggregated from the leaves up to the root of the tree in a verifiable fashion. This technique permits a wide range of efficient, tamper-evident queries, as well as enabling verifiable, safe deletion of “expired” events from the log.

Our prototype implementation supports thousands of events per second, and can easily scale to very large logs. We also demonstrated the effectiveness of Bloom filters to enable a broad range of queries. By virtue of its concise proofs and scalable design, our techniques can be applied in a variety of domains where high volumes of logged events might otherwise preclude the use of tamper-evident logs.

Acknowledgements

The authors gratefully acknowledge Farinaz Koushanfar, Daniel Sandler, and Moshe Vardi for many helpful comments and discussions on this project. The authors also thank the anonymous referees and Micah Sherr, our shepherd, for their assistance. This work was supported, in part, by NSF grants CNS-0524211 and CNS-0509297.

References

- [1] ACCORSI, R., AND HOHL, A. Delegating secure logging in pervasive computing systems. In *Security in Pervasive Computing* (York, UK, Apr. 2006), pp. 58–72.
- [2] ANAGNOSTOPOULOS, A., GOODRICH, M. T., AND TAMASSIA, R. Persistent authenticated dictionaries and their applications. In *International Conference on*

- Information Security (ISC)* (Seoul, Korea, Dec. 2001), pp. 379–393.
- [3] BELLARE, M., AND MINER, S. K. A forward-secure digital signature scheme. In *CRYPTO '99* (Santa Barbara, CA, Aug. 1999), pp. 431–448.
 - [4] BELLARE, M., AND YEE, B. S. Forward integrity for secure audit logs. Tech. rep., University of California at San Diego, Nov. 1997.
 - [5] BENALOH, J., AND DE MARE, M. One-way accumulators: a decentralized alternative to digital signatures. In *Workshop on the Theory and Application of Cryptographic Techniques on Advances in Cryptology (EuroCrypt '93)* (Lofthus, Norway, May 1993), pp. 274–285.
 - [6] BETHENCOURT, J., BONEH, D., AND WATERS, B. Cryptographic methods for storing ballots on a voting machine. In *Network and Distributed System Security Symposium (NDSS)* (San Diego, CA, Feb. 2007).
 - [7] BLIBECH, K., AND GABILLON, A. CHRONOS: An authenticated dictionary based on skip lists for timestamping systems. In *Workshop on Secure Web Services* (Fairfax, VA, Nov. 2005), pp. 84–90.
 - [8] BLOOM, B. H. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM* 13, 7 (1970), 422–426.
 - [9] BULDAS, A., LAUD, P., LIPMAA, H., AND WILLEMSON, J. Time-stamping with binary linking schemes. In *CRYPTO '98* (Santa Barbara, CA, Aug. 1998), pp. 486–501.
 - [10] BULDAS, A., LIPMAA, H., AND SCHOENMAKERS, B. Optimally efficient accountable time-stamping. In *International Workshop on Practice and Theory in Public Key Cryptography (PKC)* (Melbourne, Victoria, Australia, Jan. 2000), pp. 293–305.
 - [11] CAMENISCH, J., AND LYSYANSKAYA, A. Dynamic accumulators and application to efficient revocation of anonymous credentials. In *CRYPTO '02* (Santa Barbara, CA, Aug. 2002), pp. 61–76.
 - [12] CHAN, H., PERRIG, A., PRZYDATEK, B., AND SONG, D. SIA: Secure information aggregation in sensor networks. *Journal Computer Security* 15, 1 (2007), 69–102.
 - [13] CHAN, H., PERRIG, A., AND SONG, D. Secure hierarchical in-network aggregation in sensor networks. In *ACM Conference on Computer and Communications Security (CCS '06)* (Alexandria, VA, Oct. 2006), pp. 278–287.
 - [14] CHOR, B., GOLDBREICH, O., KUSHILEVITZ, E., AND SUDAN, M. Private information retrieval. In *Annual Symposium on Foundations of Computer Science* (Milwaukee, WI, Oct. 1995), pp. 41–50.
 - [15] CHUN, B.-G., MANIATIS, P., SHENKER, S., AND KUBIATOWICZ, J. Attested append-only memory: Making adversaries stick to their word. In *SOSP '07* (Stevenson, WA, Oct. 2007), pp. 189–204.
 - [16] D. S. PARKER, J., POPEK, G. J., RUDISIN, G., STOUGHTON, A., WALKER, B. J., WALTON, E., CHOW, J. M., EDWARDS, D., KISER, S., AND KLINE, C. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering* 9, 3 (1983), 240–247.
 - [17] DAVIS, D., MONROSE, F., AND REITER, M. K. Time-scoped searching of encrypted audit logs. In *Information and Communications Security Conference* (Malaga, Spain, Oct. 2004), pp. 532–545.
 - [18] DEUTSCH, P. Gzip file format specification version 4.3. RFC 1952, May 1996. <http://www.ietf.org/rfc/rfc1952.txt>.
 - [19] DEVANBU, P., GERTZ, M., KWONG, A., MARTEL, C., NUCKOLLS, G., AND STUBBLEBINE, S. G. Flexible authentication of XML documents. *Journal of Computer Security* 12, 6 (2004), 841–864.
 - [20] DEVANBU, P., GERTZ, M., MARTEL, C., AND STUBBLEBINE, S. G. Authentic data publication over the internet. *Journal Computer Security* 11, 3 (2003), 291–314.
 - [21] GENNARO, R., AND ROHATGI, P. How to sign digital streams. In *CRYPTO '97* (Santa Barbara, CA, Aug. 1997), pp. 180–197.
 - [22] GERR, P. A., BABINEAU, B., AND GORDON, P. C. Compliance: The effect on information management and the storage industry. The Enterprise Storage Group, May 2003. http://searchstorage.techtarget.com/tip/0,289483,sid5_gci906152,00.html.
 - [23] GOH, E.-J. Secure indexes. Cryptology ePrint Archive, Report 2003/216, 2003. <http://eprint.iacr.org/2003/216/> See also <http://eujingoh.com/papers/secureindex/>.
 - [24] GOODRICH, M., TAMASSIA, R., AND SCHWERIN, A. Implementation of an authenticated dictionary with skip lists and commutative hashing. In *DARPA Information Survivability Conference & Exposition II (DISCEX II)* (Anaheim, CA, June 2001), pp. 68–82.
 - [25] GOODRICH, M. T., TAMASSIA, R., TRIANOPOULOS, N., AND COHEN, R. F. Authenticated data structures for graph and geometric searching. In *Topics in Cryptology, The Cryptographers' Track at the RSA Conference (CT-RSA)* (San Francisco, CA, Apr. 2003), pp. 295–313.
 - [26] GOYAL, V., PANDEY, O., SAHAI, A., AND WATERS, B. Attribute-based encryption for fine-grained access control of encrypted data. In *ACM Conference on Computer and Communications Security (CCS '06)* (Alexandria, Virginia, Oct. 2006), pp. 89–98.
 - [27] HABER, S., AND STORNETTA, W. S. How to time-stamp a digital document. In *CRYPTO '98* (Santa Barbara, CA, 1990), pp. 437–455.
 - [28] HAEBERLEN, A., KOUZNETSOV, P., AND DRUSCHEL, P. PeerReview: Practical accountability for distributed systems. In *SOSP '07* (Stevenson, WA, Oct. 2007).
 - [29] HOLT, J. E. Logcrypt: Forward security and public verification for secure audit logs. In *Australasian Workshops on Grid Computing and E-research* (Hobart, Tasmania, Australia, 2006).
 - [30] HU, L., AND EVANS, D. Secure aggregation for wireless networks. In *Symposium on Applications and the Internet Workshops (SAINT)* (Orlando, FL, July 2003), p. 384.
 - [31] ITKIS, G. Cryptographic tamper evidence. In *ACM Conference on Computer and Communications Security (CCS '03)* (Washington D.C., Oct. 2003), pp. 355–364.
 - [32] KELSEY, J., CALLAS, J., AND CLEMM, A. Signed Syslog messages. <http://tools.ietf.org/id/draft-ietf-syslog-sign-23.txt> (work in progress), Sept. 2007.
 - [33] KILTZ, E., MITYAGIN, A., PANJWANI, S., AND RAGHAVAN, B. Append-only signatures. In *International Colloquium on Automata, Languages and Programming* (Lisboa, Portugal, July 2005).
 - [34] KOCHER, P. C. On certificate revocation and validation. In *International Conference on Financial Cryptography*

- (FC '98) (Anguilla, British West Indies, Feb. 1998), pp. 172–177.
- [35] KOTLA, R., ALVISI, L., DAHLIN, M., CLEMENT, A., AND WONG, E. Zyzzyva: Speculative byzantine fault tolerance. In *SOSP '07* (Stevenson, WA, Oct. 2007), pp. 45–58.
 - [36] LI, J., KROHN, M., MAZIÈRES, D., AND SHASHA, D. Secure untrusted data repository (SUNDR). In *Operating Systems Design & Implementation (OSDI)* (San Francisco, CA, Dec. 2004).
 - [37] LIPMAA, H. On optimal hash tree traversal for interval time-stamping. In *Proceedings of the 5th International Conference on Information Security (ISC02)* (Seoul, Korea, Nov. 2002), pp. 357–371.
 - [38] LONVICK, C. The BSD Syslog protocol. RFC 3164, Aug. 2001. <http://www.ietf.org/rfc/rfc3164.txt>.
 - [39] MA, D. Practical forward secure sequential aggregate signatures. In *Proceedings of the 2008 ACM symposium on Information, computer and communications security (ASIACCS'08)* (Tokyo, Japan, Mar. 2008), pp. 341–352.
 - [40] MA, D., AND TSUDIK, G. Forward-secure sequential aggregate authentication. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy* (Oakland, CA, May 2007), IEEE Computer Society, pp. 86–91.
 - [41] MA, D., AND TSUDIK, G. A new approach to secure logging. *Transactions on Storage* 5, 1 (2009), 1–21.
 - [42] MANIATIS, P., AND BAKER, M. Enabling the archival storage of signed documents. In *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies* (Monterey, CA, 2002).
 - [43] MANIATIS, P., AND BAKER, M. Secure history preservation through timeline entanglement. In *USENIX Security Symposium* (San Francisco, CA, Aug. 2002).
 - [44] MANIATIS, P., ROUSSOPOULOS, M., GIULI, T. J., ROSENTHAL, D. S. H., AND BAKER, M. The LOCKSS peer-to-peer digital preservation system. *ACM Transactions on Computer Systems* 23, 1 (2005), 2–50.
 - [45] MANULIS, M., AND SCHWENK, J. Provably secure framework for information aggregation in sensor networks. In *Computational Science and Its Applications (ICCSA)* (Kuala Lumpur, Malaysia, Aug. 2007), pp. 603–621.
 - [46] MERKLE, R. C. A digital signature based on a conventional encryption function. In *CRYPTO '88* (1988), pp. 369–378.
 - [47] MITRA, S., HSU, W. W., AND WINSLETT, M. Trustworthy keyword search for regulatory-compliant records retention. In *International Conference on Very Large Databases (VLDB)* (Seoul, Korea, Sept. 2006), pp. 1001–1012.
 - [48] MONTEIRO, S. D. S., AND ERBACHER, R. F. Exemplifying attack identification and analysis in a novel forensically viable Syslog model. In *Workshop on Systematic Approaches to Digital Forensic Engineering* (Oakland, CA, May 2008), pp. 57–68.
 - [49] NAOR, M., AND NISSIM, K. Certificate revocation and certificate update. In *USENIX Security Symposium* (San Antonio, TX, Jan. 1998).
 - [50] OSTROVSKY, R., SAHAI, A., AND WATERS, B. Attribute-based encryption with non-monotonic access structures. In *ACM Conference on Computer and Communications Security (CCS '07)* (Alexandria, VA, Oct. 2007), pp. 195–203.
 - [51] PAVLOU, K., AND SNODGRASS, R. T. Forensic analysis of database tampering. In *ACM SIGMOD International Conference on Management of Data* (Chicago, IL, June 2006), pp. 109–120.
 - [52] PETERSON, Z. N. J., BURNS, R., ATENIESE, G., AND BONO, S. Design and implementation of verifiable audit trails for a versioning file system. In *USENIX Conference on File and Storage Technologies* (San Jose, CA, Feb. 2007).
 - [53] PUGH, W. Skip lists: A probabilistic alternative to balanced trees. In *Workshop on Algorithms and Data Structures* (1989), pp. 437–449.
 - [54] SAHAI, A., AND WATERS, B. Fuzzy identity based encryption. In *Workshop on the Theory and Application of Cryptographic Techniques on Advances in Cryptology (EuroCrypt '05)* (May 2005), vol. 3494, pp. 457 – 473.
 - [55] SANDLER, D., AND WALLACH, D. S. Casting votes in the Auditorium. In *USENIX/ACCURATE Electronic Voting Technology Workshop (EVT'07)* (Boston, MA, Aug. 2007).
 - [56] SCHNEIER, B., AND KELSEY, J. Automatic event-stream notarization using digital signatures. In *Security Protocols Workshop* (Cambridge, UK, Apr. 1996), pp. 155–169.
 - [57] SCHNEIER, B., AND KELSEY, J. Secure audit logs to support computer forensics. *ACM Transactions on Information and System Security* 1, 3 (1999).
 - [58] SION, R. Strong WORM. In *International Conference on Distributed Computing Systems* (Beijing, China, May 2008), pp. 69–76.
 - [59] SNODGRASS, R. T., YAO, S. S., AND COLLBERG, C. Tamper detection in audit logs. In *Conference on Very Large Data Bases (VLDB)* (Toronto, Canada, Aug. 2004), pp. 504–515.
 - [60] SONG, D. X., WAGNER, D., AND PERRIG, A. Practical techniques for searches on encrypted data. In *IEEE Symposium on Security and Privacy* (Berkeley, CA, May 2000), pp. 44–55.
 - [61] WATERS, B. R., BALFANZ, D., DURFEE, G., AND SMETTERS, D. K. Building an encrypted and searchable audit log. In *Network and Distributed System Security Symposium (NDSS)* (San Diego, CA, Feb. 2004).
 - [62] WEATHERSPOON, H., WELLS, C., AND KUBIATOWICZ, J. Naming and integrity: Self-verifying data in peer-to-peer systems. In *Future Directions in Distributed Computing* (2003), vol. 2584 of *Lecture Notes in Computer Science*, pp. 142–147.
 - [63] YUMEREFENDI, A. R., AND CHASE, J. S. Strong accountability for network storage. *ACM Transactions on Storage* 3, 3 (2007).
 - [64] ZHU, Q., AND HSU, W. W. Fossilized index: The linchpin of trustworthy non-alterable electronic records. In *ACM SIGMOD International Conference on Management of Data* (Baltimore, MD, June 2005), pp. 395–406.

VPriv: Protecting Privacy in Location-Based Vehicular Services

Raluca Ada Popa and Hari Balakrishnan
Massachusetts Institute of Technology
Email: {ralucap,hari}@mit.edu

Andrew J. Blumberg
Stanford University
Email: blumberg@math.stanford.edu

Abstract

A variety of location-based vehicular services are currently being woven into the national transportation infrastructure in many countries. These include usage- or congestion-based road pricing, traffic law enforcement, traffic monitoring, “pay-as-you-go” insurance, and vehicle safety systems. Although such applications promise clear benefits, there are significant potential violations of the *location privacy* of drivers under standard implementations (i.e., GPS monitoring of cars as they drive, surveillance cameras, and toll transponders).

In this paper, we develop and evaluate *VPriv*, a system that can be used by several such applications without violating the location privacy of drivers. The starting point is the observation that in many applications, some centralized server needs to compute a function of a user’s *path*—a list of time-position tuples. *VPriv* provides two components: 1) the first practical protocol to compute path functions for various kinds of tolling, speed and delay estimation, and insurance calculations in a way that does not reveal anything more than the result of the function to the server, and 2) an out-of-band enforcement mechanism using random spot checks that allows the server and application to handle misbehaving users. Our implementation and experimental evaluation of *VPriv* shows that a modest infrastructure of a few multi-core PCs can easily serve 1 million cars. Using analysis and simulation based on real vehicular data collected over one year from the CarTel project’s testbed of 27 taxis running in the Boston area, we demonstrate that *VPriv* is resistant to a range of possible attacks.

1 Introduction

Over the next few years, location-based vehicular services using a combination of in-car devices and roadside surveillance systems will become a standard feature of the transportation infrastructure in many countries. Already, there is a burgeoning array of applications

of such technology, including electronic toll collection, automated traffic law enforcement, traffic statistic collection, insurance pricing using measured driving behavior, vehicle safety systems, and so on.

These services promise substantial improvements to the efficiency of the transportation network as well as to the daily experience of drivers. Electronic toll collection reduces bottlenecks at toll plazas, and more sophisticated forms of congestion tolling and usage pricing (e.g., the London congestion tolling system [24]) reduce traffic at peak times and generate revenue for transit improvements. Although the efficacy of automated traffic enforcement (e.g., stop-light cameras) is controversial, many municipalities are exploring the possibility that it will improve compliance with traffic laws and reduce accidents. Rapid collection and analysis of traffic statistics can guide drivers to choose optimal routes and allows for rational analysis of the benefits of specific allocations of transportation investments. Some insurance companies (e.g. [21]) are now testing or even deploying “pay-as-you-go” insurance programs in which insurance premiums are adjusted using information about driving behavior collected by GPS-equipped in-car devices.

Unfortunately, along with the tremendous promise of these services come very serious threats to the *location privacy* of drivers (see Section 3 for a precise definition). For instance, some current implementations of these services involve pervasive tracking—toll transponder transmitting client/account ID, license-plate cameras, mandatory in-car GPS [32], and insurance “black boxes” that monitor location and other driving information—with the data aggregated centrally by various government and corporate entities.

Furthermore, as a pragmatic matter, the widespread deployment and adoption of traffic monitoring is greatly impaired by public concern about privacy issues. A sizable impediment to further electronic tolling penetration in the San Francisco Bay Area is the refusal of a significant minority of drivers to install the devices due to

privacy concerns [31]. Privacy worries also affect the willingness of drivers to participate in the collection of traffic statistics.

This paper proposes *VPriv*, a practical system to protect a user's locational privacy while efficiently supporting a range of location-based vehicular services. *VPriv* supports applications that compute functions over the *paths* traveled by individual cars. A path is simply a sequence of *points*, where each point has a random time-varying identifier, a timestamp, and a position. Usage-based tolling, delay and speed estimation, as well as pay-as-you-go calculations can all be computed given the paths of each driver.

VPriv has two components. The first component is an *efficient protocol for tolling and speed or delay estimation that protects the location privacy of the drivers*. This protocol, which belongs to the general family of secure multi-party computations, guarantees that a joint computation between server and client can proceed correctly without revealing the private data of the parties involved. The result is that each driver (car) is guaranteed that no other information about his paths can be inferred from the computation, other than what is revealed by the result of the computed function. The idea of using multi-party secure computation in the vehicular setting is inspired from previous work [2, 3, 30]; however, these papers use multi-party computations as a black box, relying on general reductions from the literature. Unfortunately, these are extremely slow and complex, at least three orders of magnitude slower than our implementation in our experiments (see Section 8.2), which makes them unpractical.

Our main contribution here is the first *practically efficient* design, software implementation, and experimental evaluation of multi-party secure protocols for functions computed over driving paths. Our protocols exploit the specificity of cost functions over path time-location tuples: the path functions we are interested in consist of sums of costs of tuples, and we use homomorphic encryption [29] to allow the server to compute such sums using encrypted data.

The second component of *VPriv* addresses a significant concern: *making VPriv robust to physical attacks*. Although we can prove security against “cryptographic attacks” using the mathematical properties of our protocols, it is very difficult to protect against physical attacks in this fashion (e.g., drivers turning off their devices). However, one of the interesting aspects of the problem is that the embedding in a social and physical context provides a framework for discovering misbehavior. We propose and analyze a method using sporadic random spot-checks of vehicle locations that *are* linked to the actual identity of the driver. This scheme is general and independent of the function to be computed because it checks that *the argument* (driver paths) to the

secure two-party protocol is highly likely to be correct. Our analysis shows that this goal can be achieved with a small number of such checks, making this enforcement method inexpensive and minimally invasive.

We have implemented *VPriv* in C++ (and also Javascript for a browser-based demonstration). Our measurements show that the protocol runs in 100 seconds per car on a standard computer. We estimate that 30 cores of 2.4GHz speed, connected over a 100 Megabits/s link, can easily handle 1 million cars. Thus, the infrastructure required to handle an entire state's vehicular population is relatively modest. Our code is available at <http://cartel.csail.mit.edu/#vpriv>.

2 Related work

VPriv is inspired by recent work on designing cryptographic protocols for vehicular applications [2, 3, 30]. These works also discuss using random vehicle identifiers combined with secure multi-party computation or zero-knowledge proofs to perform various vehicular computations. However, these papers employ multi-party computations as a black box, relying on general results from the literature for reducing arbitrary functions to secure protocols [34]. Such protocols tend to be very complex and slow. The state of the art “general purpose” compiler for secure function evaluation, Fairplay [26], produces implementations which run more than three orders of magnitude more slowly than the *VPriv* protocol, and scale very poorly with the number of participating drivers (see Section 8.2). Given present hardware constraints, general purpose solutions for implementing secure computations are simply not viable for this kind of application. A key contribution of this paper is to present a protocol for the *specific* class of cost functions on time-location pairs, which maintains privacy and is efficient enough to be run on practical devices and suitable for deployment.

Electronic tolling and public transit fare collection were some of the early application areas for anonymous electronic cash. Satisfactory solutions to certain classes of road-pricing problems (e.g., cordon-based tolling) can be developed using electronic cash algorithms in concert with anonymous credentials [6, 25, 1]. There has been a substantial amount of work on practical protocols for these problems so that they run efficiently on small devices (e.g., [5]). Physical attacks based on the details of the implementation and the associated bureaucratic structures remain a persistent problem, however [13]. We explicitly attempt to address such attacks in *VPriv*. Our “spot check” methodology provides a novel approach to validating user participation in the cryptographic protocols, and we prove its efficiency empirically. Furthermore, unlike *VPriv*, the electronic cash

approach is significantly less suitable for more sophisticated road pricing applications, and does not apply at all to the broader class of vehicular location-based services such as “pay-as-you-go” insurance, automated traffic law enforcement, and aggregate traffic statistic collection.

There has also been a great deal of related work on protecting location privacy and anonymity while collecting vehicular data (e.g., traffic flow data) [18, 22, 16]. The focus of this work is different from ours, although it can be used in conjunction. It analyzes potential privacy violations associated with the side channels present in anonymized location databases (e.g., they conclude that it is possible to infer to what driver some GPS traces belong in regions of low density).

Using spatial analogues of the notion of *k-anonymity* [33], some work focused on using a trusted server to spatially and temporally distort locational services [15, 10]. In addition, there has been a good deal of work on using a trusted server to distort or degrade data before releasing it. An interesting class of solutions to these problems were presented in the papers [19, 17], involving “cloaking” the data using spatial and temporal subsampling techniques. In addition, these papers [17, 19] developed tools to quantify the degree of mixing of cars on a road needed to assure anonymity (notably the “time to confusion” metric). However, these solutions treat a different problem than VPriv, because most of them assume a trusted server and a non-adversarial setting, in which the user and server do not deviate from the protocol, unlike in the case of tolling or law enforcement. Furthermore, for many of the protocols we are interested in, it is not always possible to provide time-location tuples for only a subset of the space.

Nonetheless, the work in these papers complements our protocol nicely. Since VPriv does produce an anonymized location database, the analysis in [17] about designing “path upload” points that adequately preserve privacy provides a method for placing tolling regions and “spot checks” which do not violate the location privacy of users. See Section 9 for further discussion of this point.

3 Model

In this section, we describe the framework underlying our scheme, goals, and threat model. The framework captures a broad class of vehicular location-based services.

3.1 Framework

The participants in the system are *drivers*, *cars* and a *server*. Drivers operate cars, cars are equipped with transponders that transmit information to the server, and

drivers also run *client software* which enacts the cryptographic protocol on their behalf.

For any given problem (tolling, traffic statistics estimation, insurance calculations, etc.), there is one logical server and many drivers with their cars. The server computes some function f for any given car; f takes the path of the car generated during an *interaction interval* as its argument. To compute f , the server must collect the set of points corresponding to the path traveled by the car during the desired interaction interval. Each point is a tuple with three fields: $\langle \text{tag}, \text{time}, \text{location} \rangle$.

While driving, each car’s transponder generates a collection of such tuples and sends them to the server. The server computes f using the set of $\langle \text{time}, \text{location} \rangle$ pairs. If location privacy were not a concern, the *tag* could uniquely identify the car. In such a case, the server could aggregate all the tuples having the same tag and know the path of the car. Thus, in our case, these tags will be chosen at random so that they cannot be connected to an individual car. However, the driver’s client application will give the server a *cryptographic commitment* to these tags (described in Sections 4.1, 5): in our protocol, this commitment binds the driver to the particular tags and hence the result of f (e.g., the tolling cost) without revealing the tags to the server.

We are interested in developing protocols that preserve location privacy for three important functions:

1. *Usage-based tolls*: The server assesses a path-dependent toll on the car. The toll is some function of the time and positions of the car, known to both the driver and server. For example, we might have a toll that sets a particular price per mile on any given road, changing that price with time of day. We call this form of tolling a *path toll*; VPriv also supports a *point toll*, where a toll is charged whenever a vehicle goes past a certain point.
2. *Automated speeding tickets*: The server detects violations of speed restrictions: for instance, did the car ever travel at greater than 65 MPH? More generally, the server may wish to detect violations of speed limits which vary across roads and are time-dependent.
3. *“Pay-as-you-go” insurance premiums*: The server computes a “safety score” based on the car’s path to determine insurance premiums. Specifically, the server computes some function of the time, positions, and speed of the car. For example, we might wish to assess higher premiums on cars that persistently drive close to the speed limit, or are operated predominantly late at night.

These applications can be treated as essentially similar examples of the basic problem of computing a localized cost function of the car’s path represented as points. By localized we mean that the function can be decom-

posed as a sum of costs associated to a specific point or small number of specific points that are close together in space-time. In fact, our general framework can be applied to any function over path tuples because of the general result that every polynomially computable function has a secure multi-party protocol [12, 34]. However, as discussed in Section 8.2, these general results lead to impractical implementations: instead, we devise efficient protocols by exploiting the specific form of the cost functions.

In our model, each car’s transponder (transponder may be tampered with) obtains the point tuples as it drives and delivers them to the server. These tasks can be performed in several ways, depending on the infrastructure and resources available. For example, tuples can be generated as follows:

- A *GPS* device provides location and time, and the car’s transponder prepares the tuples.
- *Roadside devices* sense passing cars, communicate with a car’s transponder to receive a tag, and create a tuple by attaching time information and the fixed location of the roadside device.

Each car generates tuples periodically; depending on the specific application, either at random intervals (e.g., roughly every 30 seconds) or potentially based on location as well, for example at each intersection if the car has GPS capability. The tuples can be delivered rapidly (e.g., via roadside devices, the cellular network, or available WiFi [9]) or they can be batched until the end of the day or of the month. Section 9 describes how to avoid leaking private information when transmitting such packets to the server.

Our protocol is independent of the way these tuples are created and sent to the server, requiring only that tuples need to reach the server before the function computation. This abstract model is flexible and covers many practical systems, including in-car device systems (such as Car-Tel [20]), toll transponder systems such as E-ZPass [14], and roadside surveillance systems.

3.2 Threat model

Many of the applications of VPriv are adversarial, in that both the driver and the operator of the server may have strong financial incentives to misbehave. VPriv is designed to resist five types of attacks:

1. The driver attempts to cheat by using a modified client application during the function computation protocol to change the result of the function.
2. The driver attempts to cheat physically, by having the car’s transponder upload incorrect tuples (providing incorrect inputs to the function computation protocol):

- (a) The driver turns off or selectively disables the in-car transponder, so the car uploads no data or only a subset of the actual path data.
 - (b) The transponder uploads synthetic data.
 - (c) The transponder eavesdrops on another car and attempts to masquerade as that car.
3. The server guesses the path of the car from the uploaded tuples.
 4. The server attempts to cheat during the function computation protocol to change the result of the function or obtain information about the path of the car.
 5. Some intermediate router synthesizes false packets or systematically changes packets between the car’s transponder and the server.

All these attacks are counteracted in our scheme as discussed in Section 9. Note however that in the main discussion of the protocol, for ease of exposition we treat the server as a passive adversary; we assume that the server attempts to violate the privacy of the driver by inferring private data but correctly implements the protocol (e.g. does not claim the driver failed a verification test, when she did not). We believe this is a reasonable assumption since the server is likely to belong to an organization (e.g., the government or an insurance company) which is unlikely to engage in active attacks. However, as we discuss in Section 9, the protocol can be made resilient to a fully malicious server as well with very few modifications.

3.3 Design goals

We have the following goals for the protocol between the driver and the server, which allows the server to compute a function over a private path.

Correctness. For the car C with path P_C , the server computes the correct value of $f(P_C)$.

Location privacy. We formalize our notion of location privacy in this paper as follows:

Definition 1 (Location privacy) Let

- \mathbb{S} denote the server’s database consisting of $\langle \text{tag}, \text{time}, \text{location} \rangle$ tuples.
- \mathbb{S}' denote the database generated from \mathbb{S} by removing the tag associated to each tuple: for every tuple $\langle \text{tag}, \text{location}, \text{time} \rangle \in \mathbb{S}$, there is a tuple $\langle \text{location}, \text{time} \rangle \in \mathbb{S}'$.
- C be an arbitrary car.
- \mathcal{V} denote all the information available to the server in VPriv (“the server’s view”). This comprises the information sent by C to the server while executing the protocol (including the result of the function computation) and any other information owned or computed by the server during the computation of $f(\text{path of } C)$, (which includes \mathbb{S}).

- \mathcal{V}' denote all the information contained in \mathcal{S}' , the result of applying f on C , and any other side channels present in the raw database \mathcal{S}' .

The computation of $f(\text{path of } C)$ preserves the locational privacy of C if the server's information about C 's tuples is insignificantly larger in \mathcal{V} than in \mathcal{V}' .

Here the “insignificant amount” refers to an amount of information that cannot be exploited by a computationally bounded machine. For instance, the encryption of a text typically offers some insignificant amount of information about the text. This notion can be formalized using simulators, as is standard for this kind of cryptographic guarantee. Such a mathematical definition and proof is left for an extended version of our paper.

Informally, this definition says that the privacy guarantees of VPriv are the same as those of a system in which the server stores only tag-free path points $\langle \text{time}, \text{location} \rangle$ without any identifying information and receives (from an oracle) the result of the function (without running any protocol). Note that this definition means that any side channels present in the raw data of \mathcal{S} itself will remain in our protocols; for instance, if one somehow knows that only a single car drives on certain roads at a particular time, then that car's privacy will be violated. See Section 9 for further discussion of this issue.

Efficiency. The protocol must be sufficiently efficient so as to be feasible to run on inexpensive in-car devices. This goal can be hard to achieve; modern cryptographic protocols can be computationally intensive.

Note that we do not aim to hide the result of the function; rather, we want to compute this result without revealing private information. In some cases, such as tolling, the result may reveal information about the path of the driver. For example, a certain toll cost may be possible only by a combination of certain items. However, if the toll period is large enough, there may be multiple combinations of tolls that add to the result. Also, finding such a combination is equivalent to the subset-sum problem, which is NP-complete.

4 Architecture

This section gives an overview of the VPriv system and its components. There are three software components: the client application, which runs on the client's computer, a transponder device attached to the car, and the server software attached to a tuple database. The only requirements on the transponder are that it store a list of random tags and generate tuples as described in Section 3.1. The client application is generally assumed to be executed on the driver's home computer or mobile device like a smart-phone.

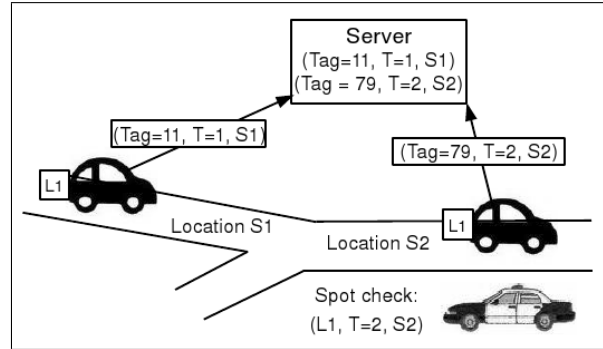


Figure 1: Driving phase overview: A car with license plate L1 is traveling from Location S1 at time 1 to Location S2 at time 2 when it undergoes a spot check. It uploads path tuples to the server.

The protocol consists of the following phases:

1. Registration. From time to time—say, upon renewing a car's registration or driver license—the driver must identify herself to the server by presenting a license or registration information. At that time, the client application generates a set of random tags that will be used in the protocol. We assume that these are indistinguishable from random by a computationally bounded adversary. The tags are also transferred to the car's transponder, but *not* given to the server. The client application then cryptographically produces *commitments* to these random tags. We describe the details of computing these commitments in Sections 4.1 and 5. The client application will provide the ciphertext of the commitments to the server and these *will* be bound to the driver's identity; however, they do not reveal any information about the actual tags under cryptographic assumptions.

2. Driving. As the car is driven, the transponder gathers time-location tuples and uploads them to the server. Each path tuple is unique because the random tag is never reused (or reused only in a precisely constrained fashion, see Section 5). The server *does not know* which car uploaded a certain tuple. To ensure that the transponder abides by the protocol, VPriv also uses sporadic random spot checks that observe the physical locations of cars, as described in Section 6. At a high level, this process generates tuples consisting of the actual license plate number, time, and location of observation. Since these spot checks record license plate information, the server knows which car they belong to. During the next phase, the client application will have to prove that the tuples uploaded by the car's transponder are consistent with these spot checks. Figure 1 illustrates the driving phase.

3. Reconciliation. This stage happens at the end of each interaction interval (e.g., at the end of the month, when a driver pays a tolling bill) and computes the function f . The client authenticates itself via a web con-

nection to the server. He does not need to transfer any information from the transponder to the computer (unless the tuples can be corrupted or lost on their way to the server and the client needs to check that they are all there). It is enough if his computer knows the initial tags (from registration). If the car had undergone a spot check, the client application has to prove that the tuples uploaded are consistent with the spot checks before proceeding (as explained in Section 6). Then, the client application initiates the function computation. The server has received tuples from the driver's car, generated in the driving phase. However, the server has also received similar tuples from many other cars and *does not know* which ones belong to a specific car. Based on this server database of tuples as well as the driver's commitment information from registration, the server and the client application conduct a cryptographic protocol in which:

- The client computes the desired function on the car's path, the path being the *private input*.
- Using a zero-knowledge proof, the client application proves to the server that the result of the function is correct, by answering correctly a series of challenges posed by the server *without revealing the driver's tags*.

The reconciliation can be done transparently to the user the client software; from the perspective of the user, he only needs to perform an online payment.

To implement this protocol, VPriv uses a set of modern cryptographic tools: a homomorphic commitment scheme and random function families. We provide a brief overview of these tools below. The experienced reader may skip to Section 5, where we provide efficient realizations that exploit details of our restricted problem setting.

4.1 Overview of cryptographic mechanisms

A **commitment scheme** [4] consists of two algorithms, *Commit* and *Reveal* or *Decommit*. Assume that Alice wants to commit to a value v to Bob. In general terms, Alice wants to provide a ciphertext to Bob from which he cannot gain any information about v . However, Alice needs to be bound to the value of v . This means that, later when she wants to reveal v to Bob, she cannot provide a different value, $v' \neq v$, which matches the same ciphertext. Specifically, she computes $Commit(v) \rightarrow (c(v), d(v))$, where $c(v)$ is the resulting ciphertext and $d(v)$ is a decommitment key with the following properties:

- Bob cannot feasibly gain any information from c .
- Alice cannot feasibly provide $v' \neq v$ such that $Commit(v') \rightarrow (c(v), d')$, for some d' .

$COST$	Path tolling cost computed by the client and reported to the server.
$c(x), d(x)$	The ciphertext and decommitment value resulting from committing to value x . That is, $Commit(x) = (c(x), d(x))$.
v_i	The random tags used by the vehicle's transponder. A subset of these will be used while driving.
(s_i, t_i)	A pair formed of a random tag uploaded at the server and the toll cost the server associates with it. $\{s_i\}$ is the set of all random tags the server received within a tolling period with $t_i > 0$.

Figure 2: Notation.

We say that Alice reveals v to Bob if she provides v and $d(v)$, the decommitment value, to Bob, who already holds $c(v)$. Note that c and d are not functions applied to v ; they are values resulting when computing $Commit(v)$ and stored for when v is revealed.

We use a **homomorphic commitment** scheme (such as the one introduced by Pedersen [29]), in which performing an arithmetic operation on the ciphertexts corresponds to some arithmetic operation on the plaintext. For instance, a commitment scheme that has the property that $c(v) \cdot c(v') = c(v + v')$ is homomorphic. Here, the decommitment key of the sum of the plaintexts is the sum of the decommitment keys $d(v + v') = d(v) + d(v')$.

A **secure multi-party computation** [34] is a protocol in which several parties hold private data and engage in a protocol in which they compute the result of a function on their private data. At the end of the protocol, the correct result is obtained and none of the participants can learn the private information of any other beyond what can be inferred from the result of the function. In this paper, we designed a variant of a secure two-party protocol. One party is the car/driver whose private data is the driving path, and the other is the server, which has no private data. A **zero-knowledge proof** [12] is a related concept that involves proving the truth of a statement without revealing any other information.

A **pseudorandom function family** [27] is a collection of functions $\{f_k\} : D \rightarrow R$ with domain D and range R , indexed by k . If one chooses k at random, for all $v \in D$, $f_k(v)$ can be computed efficiently (that is, in polynomial time) and f_k is indistinguishable from a function with random output for each input.

5 Protocols

This section presents a detailed description of the specific interactive protocol for our applications, making precise

the preceding informal description. For concreteness, we describe the protocol first in the case of the tolling application; the minor variations necessary to implement the speeding ticket and insurance premium applications are presented subsequently.

5.1 Tolling protocol

We first introduce the notation in Figure 2. For clarity, we present the protocol in a schematic manner in Figure 3. For simplicity, the protocol is illustrated for only one round. For multiple rounds, we need a *different random function for each round*. (The reason is that if the same random function is used across rounds, the server could guess the tuples of the driver by posing a $b = 0$ and a $b = 1$ challenge.) The registration phase is the same for multiple rounds, with the exception that multiple random functions are chosen in Step (a) and Steps (b) and (c) are executed for each random function.

This protocol is a case of two party-secure computation (the car is a malicious party with private data and the server is an honest but curious party) that takes the form of zero-knowledge proof: the car first computes the tolling cost and then it proves to the server that the result is correct. Intuitively, the idea of the protocol is that the client provides the server an encrypted version of her tags on which the server can compute the tolling cost in ciphertext. The server has a way of verifying that the ciphertext provided by the client is correct. The privacy property comes from the fact that the server can perform only one of the two operations at the same time: either check that the ciphertext is computed correctly, or compute the tolling cost on the vehicle tags using the ciphertext. Performing both means figuring out the driver's tuples.

These verifications and computations occur within a round, and there are multiple rounds. During each round, the server has a probability of at least $1/2$ to detect whether the client provided an incorrect $COST$, as argued in the proof below. The round protocol should be repeated s times, until the server has enough confidence in the correctness of the result. After s rounds, the probability of detecting a misbehaving client is at least $1 - (1/2)^s$, which decreases exponentially. Thus, for $s = 10$, the client is detected with 99.9% probability. The number of rounds is fixed and during registration the client selects a pseudorandom function f_k for each round and provides a set of commitments for each round.

Note that this protocol also reveals the number of tolling tuples of the car because the server knows the size of the intersection (i.e. the number of matching encryptions $f_k(v_i) = f_k(s_j)$ in iv) for $b = 1$). We do not regard this as a significant problem, since the very fact that a particular amount was paid may reveal this num-

ber (especially for cases where the tolls are about equal). However, if desired, we can handle this problem by uploading some "junk tuples". These tuples still use valid driver tags, but the location or time can be an indication to the server that they are junk and thus the server assigns a zero cost. These tuples will be included in the tolling protocol when the server will see them encrypted and will not know how many junk tuples are in the intersection of server and driver tuples and thus will not know how many actual tolling tuples the driver has. Further details of this scheme are not treated here due to space considerations.

First, it is clear that if the client is honest, the server will accept the tolling cost.

Theorem 1 *If the server responds with "ACCEPT", the protocol in Figure 3 results in the correct tolling cost and respects the driver's location privacy.*

Proof: Assume that the client has provided an incorrect tolling cost in step 3b. Note first that all decommitment keys provided to the server must be correct; otherwise the server would have detected this when checking that the commitment was computed correctly. Then, at least one of the following data provided by the client provides has to be incorrect:

- The encryption of the pairs (s_j, t_j) obtained from the server. For instance, the car could have removed some entries with high cost so that the server computes a lower total cost in step iv).
- The computation of the total toll $COST$. That is, $COST \neq \sum_{v_i=s_j} t_j$. For example, the car may have reported a smaller cost.

For if both are correct, the tolling cost computed must be correct.

During each round, the server chooses to test one of these two conditions with a probability of $1/2$. Thus, if the tolling cost is incorrect, the server will detect the misbehavior with a probability of at least $1/2$. As discussed, the detection probability increases exponentially in the number of rounds.

For location privacy, we prove that the server gains no significant additional information about the car's data other than the tolling cost and the number of tuples involved in the cost (and see above for how to avoid the latter). Let us examine the information the server receives from the client:

Step (1c): The commitments $c(k)$ and $c(f_k(v_i))$ do not reveal information by the definition of a commitment scheme.

Step (i): $c(t_j)$ does not reveal information by the definition of a commitment scheme. By the definition of the pseudorandom function, $f_k(s_i)$ looks random. After

1. Registration phase:

- Each client chooses random vehicle tags, v_i , and a random function, f_k (one per round), by choosing k at random.
- Encrypts the selected vehicle tags by computing $f_k(v_i), \forall i$, commits to the random function by computing $c(k)$, commits to the encrypted vehicle tags by computing $c(f_k(v_i))$, and stores the associated decommitment keys, $(d(k), d(f_k(v_i)))$.
- Send $c(k)$ and $c(f_k(v_i)), \forall i$ to the server. This will prevent the car from using different tags.

2. **Driving phase:** The car produces path tuples using the random tags, v_i , and sends them to the server.

3. Reconciliation phase:

- The server computes the associated tolling cost, t_j , for each random tag s_j received at the server in the last period based on the location and time where it was observed and sends (s_j, t_j) to the client only if $t_j > 0$.
- The client computes the tolling cost $COST = \sum_{v_i=s_j} t_j$ and sends it to the server.
- The **round protocol** (client proves that $COST$ is correct) begins:

Client	Server
<p>(i) Shuffle at random the pairs (s_j, t_j) obtained from the server. Encrypt s_j according to the chosen f_k random function by computing $f_k(s_j), \forall j$. Compute $c(t_j)$ and store the associated decommitments.</p> <p style="text-align: right;">Send to server $f_k(s_j)$ and $c(t_j), \forall j \rightarrow$</p> <p>(iii) If $b = 0$, the client sends k and the set of (s_j, t_j) in the shuffled order to the server and proves that these are the values she committed to in step (i) by providing $d(k)$ and $d(t_j)$. If $b = 1$, the client sends the ciphertexts of all $v_i (f_k(v_i))$ and proves that these are the values she committed to during registration by providing $d(f_k(v_i))$. The client also computes the intersection of her and the server's tags, $I = \{v_i, \forall i\} \cap \{s_j, \forall j\}$. Let $T = \{t_j : s_j \in I\}$ be the set of associated tolls to s_j in the intersection. Note that $\sum_T t_j$ represents the total tolling cost the client has to pay. By the homomorphic property discussed in Section 4.1, the product of the commitments to these tolls $t_j, \prod_{t_j \in T} c(t_j)$, is a ciphertext of the total tolling cost whose decommitment key is $D = \sum_{t_j \in T} d(t_j)$. The server will compute the sum of these costs in ciphertext in order to verify that $COST$ is correct; the client needs to provide D for this verification.</p> <p style="text-align: right;">If $b = 0, d(k), d(t_i)$ else $D, d(f_k(v_i)) \rightarrow$</p>	<p>(ii) The server picks a bit b at random. If $b = 0$, challenge the client to verify that the ciphertext provided is correct; else ($b = 1$) challenge the client to verify that the total cost based on the received ciphertext matches $COST$.</p> <p style="text-align: right;">\leftarrow Challenge random bit b</p> <p>(iv) If $b = 0$, the server verifies that all pairs (s_j, t_j) have been correctly shuffled, encrypted with f_k, and committed. This verifies that the client computed the ciphertext correctly. If $b = 1$, the server computes $\prod_{j: \exists i, f_k(v_i)=f_k(s_j)} c(t_j)$. As discussed, this yields a ciphertext of the total tolling cost and the server verifies if it is a commitment to $COST$ using D. If all checks succeed, the server <i>accepts</i> the tolling cost, else it <i>denies</i> it.</p>

Figure 3: VPriv's protocol for computing the path tolling cost (small modifications of this basic protocol work for the other applications). The arrows indicate data flow.

the client shuffles at random the pairs (s_j, t_j) , the server cannot tell which $f_k(s_j)$ corresponds to which s_j . Without such shuffling, even if the s_j is encrypted, the server would still know that the j -th ciphertext corresponds to the j -th plaintext. This will break privacy in Step (iv) for $b = 1$ when the server compares the ciphertext of s_j to the ciphertext of v_j .

Step (iii): If $b = 0$, the client will reveal k and t_j and

no further information from the client will be sent to the server in this round. Thus, the values of $f_k(v_i)$ remain committed so the server has no other information about v_i other than these committed values, which do not leak information. If $b = 1$, the client reveals $f_k(v_i)$. However, since k is not revealed, the server does not know which pseudorandom function was used and due to the pseudorandom function property, the server cannot find

v_i . Providing D only provides decommitment to the sum of the tolls which is the result of the function, and no additional information is leaked (i.e., in the case of the Pedersen scheme).

Information across rounds: A different pseudorandom function is used during every round so the information from one round cannot be used in the next round. Furthermore, the commitment to the same value in different rounds will be different and look random.

Therefore, we support our definition of location privacy because the road pricing protocol does not leak any additional information about whom the tuple tags belong to and the cars generated the tags randomly. \square

The protocol is linear in the number of tuples the car commits to during registration and the number of tuples received from the server in step 3a. It is easy to modify slightly the protocol to reduce the number of tuples that need to be downloaded as discussed in Section 7.

Point tolls (replacement of tollbooths). The predominant existing method of assessing road tolls comes from point-tolling; in such schemes, tolls are assessed at particular points, or linked to entrance/exit pairs. The latter is commonly used to charge for distance traveled on public highways. Such tolling schemes are easily handled by our protocol; tuples are generated corresponding to the tolling points. Toll that depend on the entrance/exit pairs can be handled by uploading a pair of tuples with the same tag; we discuss this refinement in detail for computation of speed below in Section 5.2. The tolling points can be “virtual”, or alternatively an implementation can utilize the existing E-Zpass infrastructure:

- The transponder knows a list of places where tuples need to be generated, or simply generates a tuple per intersection using GPS information.
- An (existing) roadside router infrastructure at tolling places can signal cars when to generate tuples.

Other tolls. Another useful toll function is charging cars for driving in certain regions. For example, cars can be charged for driving in the lower Manhattan core, which is frequently congested. One can modify the tolling cost protocol such that the server assigns a cost of 1 to every tuple inside the perimeter of this region. If the result of the function is positive, it means that the client was in the specific region.

5.2 Speeding tickets

In this application, we wish to detect and charge a driver who travels above some fixed speed limit L . For simplicity, we will initially assume that the speed limit is the same for all roads, but it is straightforward to extend the solution to varying speed limits. this constraint. The idea is to cast speed detection as a tolling problem, as follows.

We modify the driving phase to require that the car uses each random vehicle tag v_i twice; thus the car will upload pairs of linked path tuples. The server can compute the speed from a pair of linked tuples, and so during the reconciliation phase, the server assigns a cost t_i to each linked pair: if the speed computed from the pair is $> L$, the cost is non-zero, else it is zero. Now the reconciliation phase proceeds as discussed above. The spot check challenge during the reconciliation phase now requires verification that a consistent pair of tuples was generated, but is otherwise the same. If it deemed useful that the car reveal information about *where* the speeding violation occurred, the server can set the cost t_i for a violating pair to be a unique identifier for that speeding incident.

Note that this protocol leaves “gaps” in coverage during which speeding violations are not detected. Since these occur every other upload period, it is hard to imagine a realistic driver exploiting this. Likely, the driver will be travelling over the speed limit for the duration of several tuple creations. However, if this is deemed to be a concern for a given application, a variant can be used in which the period of changing tuples is divided and linked pairs are interleaved so that the whole time range is covered: $\dots v_2 v_1 v_3 v_2 v_4 v_3 v_5 v_4 v_6 v_5 \dots$

The computational costs of this protocol are analogous to the costs of the tolling protocol and so the experimental analysis of that protocol applies in this case as well. There is a potential concern about additional side channels in the server’s database associated with the use of linked tuples. Although the driver has the same guarantees as in the tolling application that her participation in the protocol does not reveal any information beyond the value of the function, the server has additional raw information in the form of the linkage. The positional information leaked in the linked tuple model is roughly the same as in the tolling model with twice the time interval between successive path tuples. Varying speed limits on different roads can be accommodated by having the prices t_i incorporate location.

5.3 Insurance premium computation

In this application, we wish to assign a “safety score” to a driver based on some function of their path which assesses their accident risk for purposes of setting insurance premiums. For example, the safety score might reflect the fraction of total driving time that is spent driving above 45 MPH at night. Or the safety score might be a count of incidents of violation of local speed limits.

As in the speeding ticket example, it is straightforward to compute these sorts of quantities from the variant of the protocol in which we require repeated use of a vehicle identifier v_i on successive tuples. If only a function

of speed and position is required, in fact the exact framework of the speeding ticket example will suffice.

6 Enforcement

The cryptographic protocol described in Section 5 ensures that a driver cannot lie about the result of the function to be computed given some private inputs to the function (the path tuples). However, when implementing such a protocol in a real setting, we need to ensure that the inputs to the function are correct. For example, the driver can turn off the transponder device on a toll road. The server will have no path tuples from that car on this road. The driver can then successfully participate in the protocol and compute the tolling cost only for the roads where the transponder was on and prove to the server that the cost was “correct”.

In this section, we present a general enforcement scheme that deals with security problems of this nature. The enforcement scheme applies to any function computed over a car’s path data.

The enforcement scheme needs to be able to detect a variety of driver misbehaviors such as using tags other than the ones committed to during registration, sending incorrect path tuples by modifying the time and location fields, failing to send path tuples, etc. To this end, we employ an end-to-end approach using sporadic *random spot checks*. We assume that at random places on the road, unknown to the drivers, there will be physical observations of a path tuple $\langle \text{license plate}, \text{time}, \text{location} \rangle$. We show in Section 8 that such spot checks can be infrequent (and thus do not affect driver privacy), while being effective.

The essential point is that the spot check tuples are connected to the car’s physical identifier, the license plate. For instance, such a spot check could be performed by secret cameras that are able to take pictures of the license plates. At the end of the day or month, an officer could extract license plate, time and location information or this task could be automated. Alternatively, using the existing surveillance infrastructure, spot checks can be carried out by roving police cars that secretly record the car information. This is similar to today’s “speed traps” and the detection probability should be the same for the same number of spot checks.

The data from the spot check is then used to validate the entries in the server database. In the reconciliation phase of the protocol from Section 5, the driver is also required to prove that she uploaded a tuple that is sufficiently close to the one observed during the spot check (and verify that the tag used in this tuple was one of the tags committed to during registration). Precisely, given a spot check tuple (t_c, ℓ_c) , the driver must prove she generated a tuple (t, ℓ) such that $|t - t_c| < \Omega_1$ and

$|\ell - \ell_c| < (\Omega_2)|t - t_c|$, where Ω_1 is a threshold related to the tuple production frequency and Ω_2 is a threshold related to the maximum rate of travel.

This proof can be performed in zero knowledge, although since the spot check reveals the car’s location at that point, this is not necessary. The driver can just present as a proof the tuple it uploaded at that location. If the driver did not upload such a tuple at the server around the observation time and place, she will not be able to claim that another driver’s tuple belongs to his due to the commitment check. The server may allow a threshold number of tuples to be missing in the database to make up for accidental errors. Before starting the protocol, a driver can check if all his tuples were received at the server and upload any missing ones.

Intuitively, we consider that the risk of being caught tampering with the protocol is akin to the current risk of being caught driving without a license plate or speeding. It is also from this perspective that we regard the privacy violation associated with the spot check method: the augmented protocol by construction reveals the location of the car at the spot check points. However, as we will show in Section 8, the number of spot checks needed to detect misbehaving drivers with high probability is very small. This means that the privacy violation is limited, and the burden on the server (or rather, whoever runs the server) of doing the spot checks is manageable.

The spot check enforcement is feasible for organizations that can afford widespread deployment of such spot checks; in practice, this would be restricted principally to governmental entities. For some applications such as insurance protocols, this assumption is unrealistic (although depending on the nature of insurance regulation in the region in question it may be the case that insurance companies could benefit from governmental infrastructure).

In this case, the protocol can be enforced by requiring auditable tamper-evident transponders. The transponder should run correctly the driving phase with tuples from registration. Correctness during the reconciliation phase is ensured by the cryptographic protocol. The insurance company can periodically check if the transponder has been tampered with (and penalize the driver if necessary). To handle the fact that the driver can temporarily disable or remove the transponder, the insurance company can check the mileage recorded by the transponder against that of the odometer, for example during annual state inspections.

7 Implementation

We implemented the road pricing protocol in C++ (577 lines on the server side and 582 on the client side). It consists of two modules, the client and the server. The

source code is available at <http://cartel.csail.mit.edu/#vpriv>. We implemented the tolling protocol from Figure 3, where we used the Pedersen commitment scheme [29] and the random function family in [27], and a typical security parameter (key size) of 128 bits (for more security, one could use a larger key size although considering the large number of commitments produced by the client, breaking a significant fraction of them is unlikely). The implementation runs the registration and reconciliation phases one after the other for one client and the server. Note that the protocol for each client is independent of the one for any other client so a logical server (which can be formed of multi-core or multiple commodity machines) could run the protocol for multiple clients in parallel.

7.1 Downloading a subset of the server's database

In the protocols described above, the client downloads the entire set of tags (along with their associated costs) from the server. When there are many clients and correspondingly the set of tags is large, this might impose unreasonable costs in terms of bandwidth and running time. In this section we discuss variants of the protocol in which these costs are reduced, at some loss of privacy.

Specifically, making a client's tags unknown among the tags of all users may not be necessary. For example, one might decide that a client's privacy would still be adequately protected if her tags cannot be distinguished in a collection of one thousand other clients' tags. Using this observation, we can trade off privacy for improved performance.

In the revised protocol, the client downloads only a subset of the total list of tags. For correctness, the client needs to prove that all of her tags are among the ones downloaded. Let the number of encrypted tags provided to the server during registration be n ; the first $m \leq n$ of these tags have been used in the last reconciliation period. Assume the driver informs the server of m . Any misreporting regarding m can be discovered by the enforcement scheme (because any tags committed to during registration but not included in the first m will not verify the spot check). When step (iv) is executed for $b = 1$, the server also checks that all the first m tuples are included in the set s_i ; that is $\{f_k(v_i) | i \leq m\} \in \{f_k(s_j) | \forall j\}$.

There are many ways in which the client could specify the subset of tags to download from the server. For instance, one way is to ask the server for some ranges of tags. For example, if the field of tags is between 0 and $(2^{128} - 1)/2^{128}$, and the client has a tag of value around 0.5673, she can ask for all the tuples with tags in the range $[0.5672, 0.5674]$. The client can ask for an interval

for each of her tags as well as for some junk intervals. The client's tag should be in a random position in the requested interval. Provided that the car tags are random, in an interval of length ΔI , if there are *total* tags, there will be about $\Delta I \cdot \text{total}$ tags.

Alternatively, during registration clients could be assigned random "tag subsets" which are then subsequently used to download clusters of tags; the number of clients per tag subset can be adjusted to achieve the desired efficiency/ privacy characteristics. The tag subset could be enforced by having the clients pick random tags with a certain prefix. Clients living in the same area would belong to the same tag subset. In this way, a driver's privacy comes from the fact that the server will not know whether the driver's tuples belong to him or to any other driver from that region (beyond any side information).

8 Evaluation

In this section we evaluate the protocols proposed. We first evaluate the implementation of the road pricing protocol. We then analyze the effectiveness of the enforcement scheme using theoretical analysis in Section 8.3.1 and with real data traces in Section 8.3.2.

We evaluated the C++ implementation by varying the number of random vehicle tags, the total number of tags seen at the server, and the number of rounds. In a real setting, these numbers will depend on the duration of the reconciliation period and the desired probability of detecting a misbehaving client. We pick random tags seen by the server and associate random costs with them. In our experiments, the server and the clients are located on the same computer, so network delays are not considered or evaluated. We believe that the network delay should not be an overhead because we can see that there are about two round trips per round. Also, the number of tuples downloaded by a client from the server should be reasonable because the client only downloads a subset of these tuples as discussed in Section 7. We are concerned primarily with measuring the cryptographic overhead.

8.1 Execution time

Figures 4, 5, and 6 show the performance results on a dual-core processor with 2.0 GHz and 1 GByte of RAM. Memory usage was rarely above 1%. The execution time for a challenge bit of 0 was typically twice as long as the one for a challenge type of 1. The running time reported is the total of the registration and reconciliation times for the server and client, averaged over multiple runs.

The graphs show an approximately linear dependency of the execution time on the parameters chosen. This

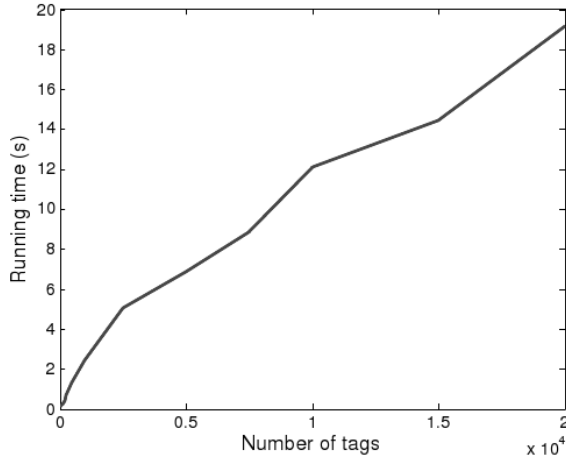


Figure 4: The running time of the road pricing protocol as a function of the number of tags generated during registration for one round.

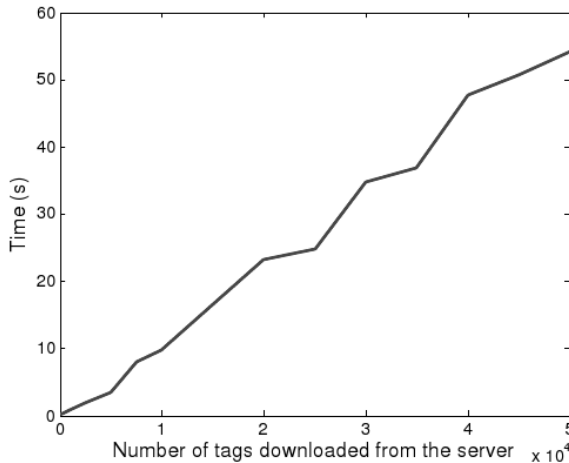


Figure 5: The running time of the road pricing protocol as a function of the number of tuples downloaded from the server during the reconciliation phase for one round.

result makes sense because all the steps of the protocol have linear complexity in these parameters.

In our experiments, we generated a random tag on average once every minute, using that tag for all the tuples collected during that minute. This interval is adjustable; the 1 minute seems reasonable given the 43 MPH average speed [28]. The average number of miles per car per year in the US is 14,500 miles and 55 min per day ([28]), which means that each month sees about ≈ 28 hours of driving per car. Picking a new tag once per minute leads to $28 \times 60 = 1680$ tags per car per month (one month is the reconciliation period that makes sense for our applications). So a car will use about 2000 tags per month.

We consider that downloading 10,000 tuples from the server offers good privacy, while increasing efficiency (note that these are only tuples with non-zero tolling cost). The reason is as follows. A person roughly drives through less than 50 toll roads per month. Assuming no side channels, the probability of guessing which tuples

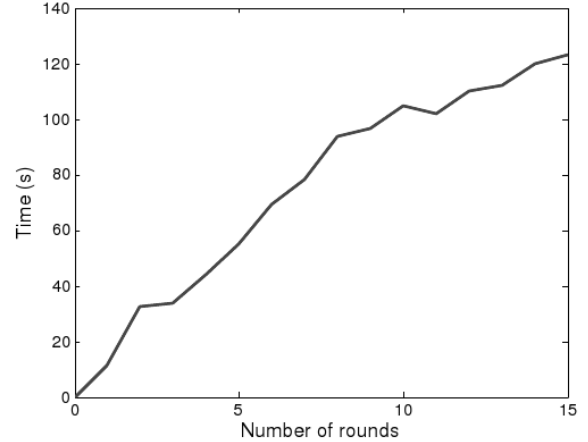


Figure 6: The running time of the road pricing protocol as a function of the number of rounds used in the protocol. The number of tags the car uses is 2000 and the number of tuples downloaded from the server is 10000.

belong to a car in this setting is $1/\binom{10000}{50}$, which is very small. Even if some of the traffic patterns of some drivers are known, the 50 tuples of the driver would be mixed in with the other 10000.

If the protocol uses 10 rounds (corresponding to a detection probability of 99.9%), the running time will be about $10 \cdot 10 = 100$ seconds, according to Figure 6. This is a very reasonable latency for a task that is done once per month and it is orders of magnitude less than the latency of the generic protocol [2] evaluated below. The server's work is typically less than half of the aggregate work, that is, 50 seconds. Downloading 10,000 tuples (each about 50 bytes) at a rate of 10Mb/s yields an additional delay of 4 seconds. Therefore, one similar core could handle 30 days per month times 86400 seconds per day divided by 54 seconds per car = 51840 cars per month. Even if bandwidth does not scale linearly with the number of cores, the latency due to bandwidth utilization is still one order of magnitude less than the one for computation; even if it adds up and cannot be parallelized, the needed number of cores is still within the same order of magnitude. Also, several computers can be placed in different parts of the network in order to parallelize the use of wide-area bandwidth. Since the downloaded content for drivers in the same area is the same, a proxy in certain regions will decrease bandwidth usage significantly. Hence, for 1 million cars, one needs $10^6/51840 \approx 21 < 30$ similar cores; this computation suggests our protocol is feasible for real deployment. (We assumed equal workloads per core because each core serves about 50000 users so the variance among cores is made small.)

8.2 Comparison to Fairplay

Fairplay [26] is a general-purpose compiler for producing secure two-party protocols that implement arbitrary functions. It generates circuits using Yao's classic work on secure two-party computation [34]. We implemented a simplified version of the tolling protocol in Fairplay. The driver has a set of tuples and the server simply computes the sum of the costs of some of these tuples. We made such simplifications because the Fairplay protocol was prohibitively slow with a more similar protocol to ours. Also, in our implementation, the Fairplay server has no private state (to match our setting in which the private state is only on the client). We found that the performance and resource consumption of Fairplay were untenable for very small-sized instances of this problem. The Fairplay program ran out of 1 GB of heap space for a server database of only 75 tags, and compiling and running the protocol in such a case required over 5 minutes. In comparison, our protocol runs with about 10,000 tuples downloaded from the server in 100s, which yields a difference in performance of *three orders of magnitude*. In addition, the oblivious circuit generated in this case was over 5 MB, and the scaling (both for memory and latency) appeared to be worse than linear in the number of tuples. There have been various refinements to aspects of Fairplay since its introduction which significantly improve its performance and bandwidth requirements; notably, the use of ordered binary decision diagrams [23]. However, the performance improvements associated with this work are less than an order of magnitude at best, and so do not substantially change the general conclusion that the general-purpose implementation of the relevant protocol is orders of magnitude slower than VPriv. This unfeasibility of using existing general frameworks required us to invent our own protocol for cost functions over path tuples that is efficient and provides the same security guarantees as the general protocols.

8.3 Enforcement effectiveness

We now analyze the effectiveness of the enforcement scheme both analytically and using trace-driven experiments. We would like to show that the time a motorist can drive illegally and the number of required spot checks are small. We will see that the probability to detect a misbehaving driver grows exponentially in the number of spot checks, making the number of spot checks logarithmic in the desired detection probability. This result is attractive from the dual perspectives of implementation cost and privacy preservation.

8.3.1 Analytical evaluation

We perform a probabilistic analysis of the time a motorist can drive illegally as well as the number of spot checks required. Let p be the probability that a driver undergoes a spot check in a one-minute interval (or similarly, driving through a segment). Let m be the number of minutes until a driver is detected with a desired probability. The number of spot checks a driver undergoes is a binomial random variable with parameters (p, m) , pm being its expected value.

The probability that a misbehaving driver undergoes at least one spot check in m minutes is

$$\Pr[\text{spot check}] = 1 - (1 - p)^m. \quad (1)$$

Figure 7 shows the number of minutes a misbehaving driver will be able to drive before it will be observed with high probability. This time decreases exponentially in the probability of a spot check in each minute. Take the example of $p = 1/500$. In this case, each car has an expected time of 500 minutes (8.3h) of driving until it undergoes a spot check and will be observed with 95% probability after about 598 min (< 10 hours) of driving, which means that overwhelmingly likely the driver will not be able to complete a driving period of a month without being detected.

However, a practical application does not need to ensure that cars upload tuples on all the roads. In the road pricing example, it is only necessary to ensure that cars upload tuples on toll roads. Since the number of toll points is usually only a fraction of all the roads, a much smaller number of spot checks will suffice. For example, if we have a spot check at one tenth of the tolling roads, after 29 minutes, each driver will undergo a spot check with 95% probability.

Furthermore, if the penalty for failing the spot check test is high, a small number of spot checks would suffice because even a small probability of detecting each driver would eliminate the incentive to cheat for many drivers. In order to ensure compliance by rational agents, we simply need to ensure that the penalty associated with noncompliance, β , is such that $\beta(\Pr[\text{penalization}]) > \alpha$, where α is the total toll that could possibly be accumulated over the time period. Of course, evidence from randomized law enforcement suggests strongly that independent of β , $\Pr[\text{penalization}]$ needs to be appreciable (that is, a driver must have confidence that they *will* be caught if they persist in flouting the compliance requirements) [8].

If there is concern about the possibility of tuples lost in transit from client to server, our protocol can be augmented with an anonymized interaction in which a client checks to see if all of her tuples are included in the server's database (the client can perform this check af-

ter downloading the desired tuples from the server and before the spot check reconciliation and zero-knowledge protocol). Alternatively, the client might simply blindly upload duplicates of all her tuples at various points throughout the month to ensure redundant inclusion in the database. Note that it is essential that this interaction should be desynchronized from the reconciliation process in order to prevent linkage and associated privacy violation.

Nevertheless, even if we allow for a threshold t of tuples to be lost before penalizing a driver, the probability of detection is still exponential in the driving time $1 - \sum_{i=0}^t \binom{m}{i} p^i (1-p)^{m-i} \geq 1 - e^{-\frac{(t-mp)^2}{2mp}}$, where the last inequality uses Chernoff bounds.

8.3.2 Experimental evaluation

We now evaluate the effectiveness of the enforcement scheme using a trace-driven experimental evaluation. We obtained real traces from the CarTel project testbed [20], containing the paths of 27 limousine drivers mostly in the Boston area, though extending to other MA, NH, RI, and CT areas, during a one-year period (2008). Each car drives many hours every day. The cars carry GPS sensors that record location and time. We match the locations against the Navteq map database. The traces consist of tuples of the form (car tag, segment tag, time) generated at intervals with a mean of 20 seconds. Each segment represents a continuous piece of road between two intersections (one road usually consists of many segments).

We model each spot check as being performed by a police car standing by the side of a road segment. The idea is to place such police cars on certain road segments, to replay the traces, and verify how many cars would be spot-checked.

We do not claim that our data is representative of the driving patterns of most motorists. However, these are the best real data traces we could obtain with driver, time, and location information. We believe that such data is still informative; one might argue that a limousine's path is an aggregation of the paths of the different individuals that took the vehicles in one day.

It is important to place spot checks randomly to prevent misbehaving drivers from knowing the location of the spot checks and consequently to behave correctly only in that area. One solution is to examine traffic patterns and to determine the most frequently travelled roads. Then, spot checks would be placed with higher probability on popular roads and with lower probability on less popular roads. This scheme may not observe a malicious client driving through very sparsely travelled places; however, such clients may spend fuel and time resources by driving through these roads and which most likely do not even have tolls. More sophisticated place-

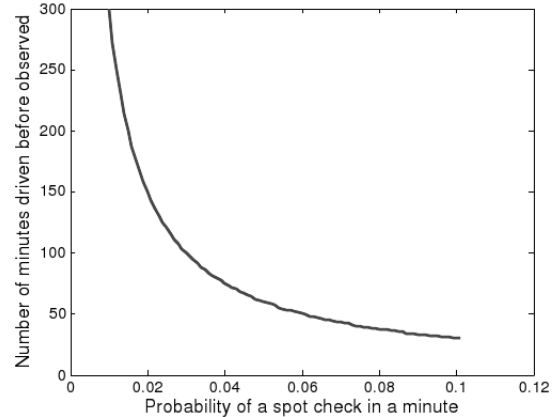


Figure 7: The time a motorist can drive illegally before it undergoes a spot check with a probability 95% for various values of p , the probability a driver undergoes a spot check in a minute.

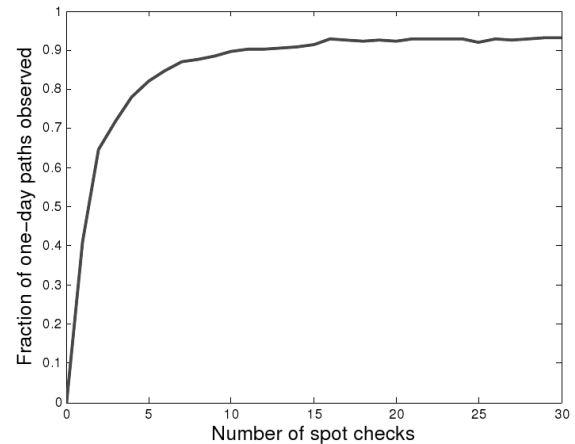


Figure 8: The fraction of one-day paths observed out of a total of 4826 one-day paths as a function of the total number of police cars placed.

ment schemes are possible; here, we are primarily concerned with showing the ability to observe most traffic with remarkably few spot checks.

Consider the following experiment: we use the traces from a month as a training phase and the traces from the next month as a testing phase, for each month except for the last one. The first month is used to determine the first 1% (≈ 300) popular sites. We choose an increasing number of police cars to be placed randomly at some of these sites. Then, in the testing phase we examine how many drivers are observed in the next month. We perform this experiment for an increasing number of police cars and for each experiment we average the results over fifty runs. In order to have a large sample, we consider the paths of a driver in two different days as the paths of two different drivers. This yields 4826 different one-day traces.

Figure 8 illustrates the data obtained. In few places, the graph is not perfectly monotonic and this is due to randomization: we are placing few spot checks in some

of the 300 locations. Even if in some cases we place a spot check more than in others, due to randomization, the spot checks may be placed in an unfavorable position and observe less paths. The reason is that the 300 spot check vary significantly in popularity. From the shape of the graph, we can see that the fraction of paths observed increases very fast at the beginning; this is explained by the exponential behavior discussed in Section 8.3.1. After 10 spot checks have been placed, the fraction of paths observed grows much slower. This is because we are only placing spot checks at 1% of the segments traveled by the limousine drivers. Some one-day paths may not be included at all in this set of paths. Overall, we can see that this algorithm requires a relatively small number of police cars, namely 20, to observe $\approx 90\%$ of the 4826 one-day paths.

Our data unfortunately does not reflect the paths of the entire population of a city and we could not find such extensive trace data. A natural question to ask would be how many police cars would be needed for a large city. We speculate that this number is larger than the number of drivers by a sublinear factor in the size of the population; according to the discussion in Section 8.3.1, the number of spot checks increases logarithmically in the probability of detection of each driver and thus the percentage of drivers observed.

9 Security analysis

In this section, we discuss the resistance of our protocol to the various attacks outlined in Section 3.2.

Client and intermediate router attacks. Provided that the client's tuples are successfully and honestly uploaded at the server, the analysis of Section 5 shows that the client cannot cheat about the result of the function. To ensure that the tuples arrive uncorrupted, the client should encrypt tuples with the public key of the server. To deal with dropped or forged tuples, the drivers should make sure that all their tuples are included in the subset of tuples downloaded from the server during the function computation. If some tuples are missing, the client can upload them to the server. These measures overcome any misbehavior on the part of intermediate routers.

The spot check method (backed with an appropriate penalty) is a strong disincentive for client misbehavior. An attractive feature of the spot check scheme is that it protects against attacks involving bad tuple uploads by drivers. For example, drivers cannot turn off their transponders because they will fail the spot check test; they will not be able to provide a consistent tuple. Similarly, drivers cannot use invalid tags (synthetic or copied from another driver), because the client will then not pass the spot checks; the driver did not commit to such tags during registration.

If two drivers agree to use the same tags (and commit

to them in registration), they will both be responsible for the result of the function (i.e., they will pay the sum of the tolling amounts for both of them).

Server misbehavior. Provided that the server honestly carries out the protocol, the analysis of Section 5 shows that it cannot obtain any additional information from the cryptographic protocol. A concern could be that the server attempts to track the tuples a car sends by using network information (e.g., IP address). Well-studied solutions from the network privacy and anonymization literature can be used here, such as Tor [7], or onion routing [11]. The client can avoid any timing coincidence by sending these tuples in separate packets (perhaps even at some intervals of time) towards the end of the driving period, when other people are sending such tuples.

Another issue is the presence of side channels in the anonymized tuple database. As discussed in Section 2, a number of papers have demonstrated that in low-density regions it is possible to reconstruct paths with some accuracy from anonymized traces [18, 22, 16]. As formalized in Definition 1, our goal in this paper was to present a protocol that avoids leaking any additional information beyond what can be deduced from the anonymized database. The obvious way to prevent this kind of attack is to restrict the protocol so that tuples are uploaded (and spot checks are conducted) only in areas of high traffic density. An excellent framework for analyzing potential privacy violations has been developed in [19, 17], which use a *time to confusion* metric that measures how long it takes an identified vehicle to mix back into traffic. In [17], this is used to design traffic information upload protocols with exclusion areas and spacing constraints so as to reduce location privacy loss.

Recall that in Section 5, we assumed that the server is a passive adversary: it is trusted not to change the result of the function, although it tries to obtain private information. A malicious server might dishonestly provide tuples to the driver or compute the function f wrongly. With a few changes to the protocol, however, VPriv can be made resilient to such attacks.

- The function f is made public. In Figure 3, step 3a), the server computes the tolls associated to each tuple. A malicious server can attach any cost to each tuple, and to counteract this, we require that the tolling function is public. Thus, the client can compute the cost of each tuple in a verifiable way.
- For all the client commitments sent to the server, the client must also provide to the server a signed hash of the ciphertext. This will prevent the server from changing the client's ciphertext because he cannot forge the client's signature.
- When the server sends the client the subset of tuples in Step 3a), the server needs to send a signed hash of these values as well. Then, the server cannot change

his mind about the tuples provided.

- The server needs to prove to a separate entity that the client misbehaved during enforcement before penalizing it (eg. insurance companies must show the tamper-evident device).

Note that it is very unlikely that the server could drop or modify the tuples of a specific driver because the server does not know which ones belong to the driver and would need to drop or modify a large, detectable number of tuples. If the server rejects the challenge information of the client in Step iv) when it is correct, then the client can prove to another person that its response to the challenge is correct.

10 Conclusion

In this paper, we presented VPriv, a practical system to protect a driver's location privacy while efficiently supporting a range of location-based vehicular services. VPriv combined cryptographic protocols to protect the location privacy of the driver with a spot check enforcement method. A central focus of our work was to ensure that VPriv satisfies pragmatic goals: we wanted VPriv to be efficient enough to run on stock hardware, to be sufficiently flexible so as to support a variety of location-based applications, to be implementable with many different physical setups, and to resist a wide array of physical attacks. We verified through analytical results and simulation using real vehicular data that VPriv realized these goals.

Acknowledgments. This work was supported in part by the National Science Foundation under grants CNS-0716273 and CNS-0520032. We thank the members of the CarTel project, especially Jakob Eriksson, Sejoon Lim, and Sam Madden for the vehicle traces, and Seth Riney of PlanetTran. David Andersen, Sharon Goldberg, Ramki Gummadi, Sachin Katti, Petros Maniatis, and the members of the PMG group at MIT have provided many useful comments on this paper. We thank Robin Chase and Roy Russell for helpful discussions.

References

- [1] BANGERTER, E., CAMENISCH, J., AND LYSYANSKAYA, A. A cryptographic framework for the controlled release of certified data. In *Security Protocols Workshop* (2004).
- [2] BLUMBERG, A., AND CHASE, R. Congestion pricing that respects "driver privacy". In *ITSC* (2005).
- [3] BLUMBERG, A., KEELER, L., AND SHELAT, A. Automated traffic enforcement which respects driver privacy. In *ITSC* (2004).
- [4] BRASSARD, G., CHAUM, D., AND CREPEAU, C. Minimum disclosure proofs of knowledge. In *JCSS*, 37, pp. 156-189 (1988).
- [5] CAMENISCH, J., HOHENBERGER, S., AND LYSYANSKAYA, A. Balancing accountability and privacy using e-cash. In *SCN* (2006).
- [6] CHAUM, D. Security without identification: transaction systems to make big brother obsolete. In *CACM* 28(10) (1985).
- [7] DINGLEDINE, R., MATHEWSON, N., AND SYVERSON, P. Tor: The second-generation onion router. In *USENIX Sec. Symp., USENIX Association* (2004).
- [8] EIDE, E., RUBIN, P. H., AND SHEPHERD, J. *Economics of crime*. Now Publishers, 2006.
- [9] ERIKSSON, J., BALAKRISHNAN, H., AND MADDEN, S. Cabernet: Vehicular content delivery using wifi. In *MOBICOM* (2008).
- [10] GEDIK, B., AND LIU, L. Location privacy in mobile systems: A personalized anonymization model. In *25th IEEE ICDCS* (2005).
- [11] GOLDSCHLAG, D., REED, M., AND SYVERSON, P. Onion routing for anonymous and private internet connections. In *CACM*, 42(2) (1999).
- [12] GOLDWASSER, S., MICALI, S., AND RACKOFF, C. The knowledge complexity of interactive proof-systems. In *Proceedings of 17th Symposium on the Theory of Computation, Providence, Rhode Island*. (1985).
- [13] GOODIN, D. Microscope-wielding boffins crack tube smartcard.
- [14] GROUP, E.-Z. I. E-zpass.
- [15] GRUTESER, M., AND GRUNWALD, D. Anonymous usage of location-based services through spatial and temporal cloaking. In *ACM MobiSys* (2003).
- [16] GRUTESER, M., AND HOH, B. On the anonymity of periodic location samples. In *Pervasive* (2005).
- [17] HOH, B., GRUTESER, M., HERRING, R., BAN, J., WORK, D., HERRERA, J.-C., BAYEN, A., ANNAVARAM, M., AND JACOBSON, Q. Virtual trip lines for distributed privacy-preserving traffic monitoring. In *Mobisys* (2008).
- [18] HOH, B., GRUTESER, M., XIONG, H., AND ALRABADY, A. Enhancing security and privacy in traf- monitoring systems. In *IEEE Pervasive Computing*, 5(4):38-46 (2006).
- [19] HOH, B., GRUTESER, M., XIONG, H., AND ALRABADY, A. Preserving privacy in gps traces via uncertainty-aware path cloaking. In *ACM CCS* (2007).
- [20] HULL, B., BYCHKOVSKY, V., CHEN, K., GORACZKO, M., MIU, A., SHIH, E., ZHANG, Y., BALAKRISHNAN, H., AND MADDEN, S. Cartel: A distributed mobile sensor computing system. In *ACM SenSys* (2006).
- [21] INSURANCE, A. Mile meter.
- [22] KRUMM, J. Inference attacks on location tracks. In *Pervasive* (2007).
- [23] L. KRUGER, E. GOH, S. J., AND BONEH, D. Secure function evaluation with ordered binary decision diagrams. In *ACM CCS* (2006).
- [24] LITMAN, T. London congestion pricing, 2006.
- [25] LYSYANSKAYA, A., RIVEST, R., SAHAI, A., , AND WOLF, S. *Pseudonym systems*. Springer, 2000.
- [26] MALKHI, D., NISAN, N., PINKAS, B., AND SELLA, Y. Fairplay - a secure two-party computation system. In *USENIX Sec. Symp., USENIX Association* (2004).
- [27] NAOR, M., AND REINGOLD, O. Number-theoretic constructions of efficient pseudo-random functions. In *Journal of the ACM, Volume 51, Issue 2, p. 231-262* (March 2004).
- [28] OF TRANSPORTATION STATISTICS, B. National household travel survey daily travel quick facts.
- [29] PEDERSEN, T. P. Non-interactive and information-theoretic secure verifiable secret sharing. In *Springer-Verlag* (1998).
- [30] RASS, S., FUCHS, S., SCHAFFER, M., AND KYAMAKYA, K. How to protect privacy in floating car data systems. In *Proceedings of the fifth ACM international workshop on VehiculAr Inter- Networking* (2008).
- [31] RILEY, P. The tolls of privacy: An underestimated roadblock for electronic toll collection usage. In *Third International Conference on Legal, Security + Privacy Issues in IT* (2008).
- [32] SALLADAY, R. Dmv chief backs tax by mile. In *Los Angeles Times* (November 16, 2004).
- [33] SWEENEY, L. k-anonymity: A model for protecting privacy. In *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems v.10 n.5* (2002).
- [34] YAO, A. C. Protocols for secure computations (extended abstract). In *FOCS* (1982: 160-164).

Effective and Efficient Malware Detection at the End Host

Clemens Kolbitsch*, Paolo Milani Comparetti*, Christopher Kruegel[‡], Engin Kirda[§],
Xiaoyong Zhou[†], and XiaoFeng Wang[†]

*Secure Systems Lab, TU Vienna

{ck, pmilani}@seclab.tuwien.ac.at

[‡]UC Santa Barbara

chris@cs.ucsb.edu

[§]Institute Eurecom, Sophia Antipolis

kirda@eurecom.fr

[†]Indiana University at Bloomington

{zhou, xw7}@indiana.edu

Abstract

Malware is one of the most serious security threats on the Internet today. In fact, most Internet problems such as spam e-mails and denial of service attacks have malware as their underlying cause. That is, computers that are compromised with malware are often networked together to form botnets, and many attacks are launched using these malicious, attacker-controlled networks.

With the increasing significance of malware in Internet attacks, much research has concentrated on developing techniques to collect, study, and mitigate malicious code. Without doubt, it is important to collect and study malware found on the Internet. However, it is even more important to develop mitigation and detection techniques based on the insights gained from the analysis work. Unfortunately, current host-based detection approaches (i.e., anti-virus software) suffer from *ineffective* detection models. These models concentrate on the features of a specific malware instance, and are often easily evadable by obfuscation or polymorphism. Also, detectors that check for the presence of a sequence of system calls exhibited by a malware instance are often evadable by system call reordering. In order to address the shortcomings of ineffective models, several dynamic detection approaches have been proposed that aim to identify the behavior exhibited by a malware family. Although promising, these approaches are unfortunately *too slow* to be used as real-time detectors on the end host, and they often require cumbersome virtual machine technology.

In this paper, we propose a novel malware detection approach that is both *effective* and *efficient*, and thus, can be used to replace or complement traditional anti-virus software at the end host. Our approach first analyzes a malware program in a controlled environment to build a model that characterizes its behavior. Such models describe the information flows between the system calls essential to the malware's mission, and therefore, cannot be easily evaded by simple obfuscation or polymorphic techniques. Then, we extract the program slices respon-

sible for such information flows. For detection, we execute these slices to match our models against the runtime behavior of an unknown program. Our experiments show that our approach can effectively detect running malicious code on an end user's host with a small overhead.

1 Introduction

Malicious code, or malware, is one of the most pressing security problems on the Internet. Today, millions of compromised web sites launch drive-by download exploits against vulnerable hosts [35]. As part of the exploit, the victim machine is typically used to download and execute malware programs. These programs are often bots that join forces and turn into a botnet. Botnets [14] are then used by miscreants to launch denial of service attacks, send spam mails, or host scam pages.

Given the malware threat and its prevalence, it is not surprising that a significant amount of previous research has focused on developing techniques to collect, study, and mitigate malicious code. For example, there have been studies that measure the size of botnets [37], the prevalence of malicious web sites [35], and the infestation of executables with spyware [31]. Also, a number of server-side [4, 43] and client-side honeypots [51] were introduced that allow analysts and researchers to gather malware samples in the wild. In addition, there exist tools that can execute unknown samples and monitor their behavior [6, 28, 54, 55]. Some tools [6, 54] provide reports that summarize the activities of unknown programs at the level of Windows API or system calls. Such reports can be evaluated to find clusters of samples that behave similarly [5, 7] or to classify the type of observed, malicious activity [39]. Other tools [55] incorporate data flow into the analysis, which results in a more comprehensive view of a program's activity in the form of taint graphs.

While it is important to collect and study malware, this is only a means to an end. In fact, it is crucial that

the insight obtained through malware analysis is translated into detection and mitigation capabilities that allow one to eliminate malicious code running on infected machines. Considerable research effort was dedicated to the extraction of network-based detection models. Such models are often manually-crafted signatures loaded into intrusion detection systems [33] or bot detectors [20]. Other models are generated automatically by finding common tokens in network streams produced by malware programs (typically, worms) [32, 41]. Finally, malware activity can be detected by spotting anomalous traffic. For example, several systems try to identify bots by looking for similar connection patterns [19, 38]. While network-based detectors are useful in practice, they suffer from a number of limitations. First, a malware program has many options to render network-based detection very difficult. The reason is that such detectors cannot observe the activity of a malicious program directly but have to rely on artifacts (the traffic) that this program produces. For example, encryption can be used to thwart content-based techniques, and blending attacks [17] can change the properties of network traffic to make it appear legitimate. Second, network-based detectors cannot identify malicious code that does not send or receive any traffic.

Host-based malware detectors have the advantage that they can observe the complete set of actions that a malware program performs. It is even possible to identify malicious code before it is executed at all. Unfortunately, current host-based detection approaches have significant shortcomings. An important problem is that many techniques rely on *ineffective* models. Ineffective models are models that do not capture intrinsic properties of a malicious program and its actions but merely pick up artifacts of a specific malware instance. As a result, they can be easily evaded. For example, traditional anti-virus (AV) programs mostly rely on file hashes and byte (or instruction) signatures [46]. Unfortunately, obfuscation techniques and code polymorphism make it straightforward to modify these features without changing the actual semantics (the behavior) of the program [10]. Another example are models that capture the sequence of system calls that a specific malware program executes. When these system calls are independent, it is easy to change their order or add irrelevant calls, thus invalidating the captured sequence.

In an effort to overcome the limitations of ineffective models, researchers have sought ways to capture the malicious activity that is characteristic of a malware program (or a family). On one hand, this has led to detectors [9, 12, 25] that use sophisticated static analysis to identify code that is semantically equivalent to a malware template. Since these techniques focus on the actual semantics of a program, it is not enough for a malware

sample to use obfuscation and polymorphic techniques to alter its appearance. The problem with static techniques is that static binary analysis is difficult [30]. This difficulty is further exacerbated by runtime packing and self-modifying code. Moreover, the analysis is costly, and thus, not suitable for replacing AV scanners that need to quickly scan large numbers of files. Dynamic analysis is an alternative approach to model malware behavior. In particular, several systems [22, 55] rely on the tracking of dynamic data flows (tainting) to characterize malicious activity in a generic fashion. While detection results are promising, these systems incur a significant performance overhead. Also, a special infrastructure (virtual machine with shadow memory) is required to keep track of the taint information. As a result, static and dynamic analysis approaches are often employed in automated malware analysis environments (for example, at anti-virus companies or by security researchers), but they are too *inefficient* to be deployed as detectors on end hosts.

In this paper, we propose a malware detection approach that is both *effective* and *efficient*, and thus, can be used to replace or complement traditional AV software at the end host. For this, we first generate effective models that cannot be easily evaded by simple obfuscation or polymorphic techniques. More precisely, we execute a malware program in a controlled environment and observe its interactions with the operating system. Based on these observations, we generate fine-grained models that capture the characteristic, malicious behavior of this program. This analysis can be expensive, as it needs to be run only once for a group of similar (or related) malware executables. The key of the proposed approach is that our models can be efficiently matched against the runtime behavior of an unknown program. This allows us to detect malicious code that exhibits behavior that has been previously associated with the activity of a certain malware strain.

The main contributions of this paper are as follows:

- We automatically generate fine-grained (effective) models that capture detailed information about the behavior exhibited by instances of a malware family. These models are built by observing a malware sample in a controlled environment.
- We have developed a scanner that can efficiently match the activity of an unknown program against our behavior models. To achieve this, we track dependencies between system calls without requiring expensive taint analysis or special support at the end host.
- We present experimental evidence that demonstrates that our approach is feasible and usable in practice.

2 System Overview

The goal of our system is to effectively and efficiently detect malicious code at the end host. Moreover, the system should be general and not incorporate *a priori* knowledge about a particular malware class. Given the freedom that malware authors have when crafting malicious code, this is a challenging problem. To attack this problem, our system operates by generating detection models based on the observation of the execution of malware programs. That is, the system executes and monitors a malware program in a controlled analysis environment. Based on this observation, it extracts the behavior that characterizes the execution of this program. The behavior is then automatically translated into detection models that operate at the host level.

Our approach allows the system to quickly detect and eliminate novel malware variants. However, it is reactive in the sense that it must observe a certain, malicious behavior before it can properly respond. This introduces a small delay between the appearance of a new malware family and the availability of appropriate detection models. We believe that this is a trade-off that is necessary for a general system that aims to detect and mitigate malicious code with *a priori* unknown behavior. In some sense, the system can be compared to the human immune system, which also reacts to threats by first detecting intruders and then building appropriate antibodies. Also, it is important to recognize that it is *not* required to observe every malware instance before it can be detected. Instead, the proposed system abstracts (to some extent) program behavior from a single, observed execution trace. This allows the detection of all malware instances that implement similar functionality.

Modeling program behavior. To model the behavior of a program and its security-relevant activity, we rely on system calls. Since system calls capture the interactions of a program with its environment, we assume that any relevant security violation is visible as one or more unintended interactions.

Of course, a significant amount of research has focused on modeling legitimate program behavior by specifying permissible sequences of system calls [18, 48]. Unfortunately, these techniques cannot be directly applied to our problem. The reason is that malware authors have a large degree of freedom in rearranging the code to achieve their goals. For example, it is very easy to reorder independent system calls or to add irrelevant calls. Thus, we cannot represent suspicious activity as system call sequences that we have observed. Instead, a more flexible representation is needed. This representation must capture true relationships between system calls but allow independent calls to appear in any order. For

this, we represent program behavior as a *behavior graph* where nodes are (interesting) system calls. An edge is introduced from a node x to node y when the system call associated with y uses as argument some output that is produced by system call x . That is, an edge represents a data dependency between system calls x and y . Moreover, we only focus on a subset of interesting system calls that can be used to carry out malicious activity.

At a conceptual level, the idea of monitoring a piece of malware and extracting a model for it bears some resemblance to previous signature generation systems [32, 41]. In both cases, malicious activity is recorded, and this activity is then used to generate detection models. In the case of signature generation systems, network packets sent by worms are compared to traffic from benign applications. The goal is to extract tokens that are unique to worm flows and, thus, can be used for network-based detection. At a closer look, however, the differences between previous work and our approach are significant. While signature generation systems extract specific, byte-level descriptions of malicious traffic (similar to virus scanners), the proposed approach targets the semantics of program executions. This requires different means to observe and model program behavior. Moreover, our techniques to identify malicious activity and then perform detection differ as well.

Making detection efficient. In principle, we can directly use the behavior graph to detect malicious activity at the end host. For this, we monitor the system calls that an unknown program issues and match these calls with nodes in the graph. When enough of the graph has been matched, we conclude that the running program exhibits behavior that is similar to previously-observed, malicious activity. At this point, the running process can be terminated and its previous, persistent modifications to the system can be undone.

Unfortunately, there is a problem with the previously sketched approach. The reason is that, for matching system calls with nodes in the behavior graph, we need to have information about data dependencies between the arguments and return values of these systems calls. Recall that an edge from node x to y indicates that there is a data flow from system call x to y . As a result, when observing x and y , it is not possible to declare a match with the behavior graph $x \rightarrow y$. Instead, we need to know whether y uses values that x has produced. Otherwise, independent system calls might trigger matches in the behavior graph, leading to an unacceptable high number of false positives.

Previous systems have proposed dynamic data flow tracking (tainting) to determine dependencies between system calls. However, tainting incurs a significant performance overhead and requires a special environ-

ment (typically, a virtual machine with shadow memory). Hence, taint-based systems are usually only deployed in analysis environments but not at end hosts. In this paper, we propose an approach that allows us to detect previously-seen data dependencies by monitoring only system calls and their arguments. This allows efficient identification of data flows without requiring expensive tainting and special environments (virtual machines).

Our key idea to determine whether there is a data flow between a pair of system calls x and y that is similar to a previously-observed data flow is as follows: Using the observed data flow, we extract those parts of the program (the instructions) that are responsible for reading the input and transforming it into the corresponding output (a kind of program slice [53]). Based on this program slice, we derive a symbolic expression that represents the semantics of the slice. In other words, we extract an expression that can essentially pre-compute the expected output, based on some input. In the simplest case, when the input is copied to the output, the symbolic expression captures the fact that the input value is identical to the output value. Of course, more complicated expressions are possible. In cases where it is not possible to determine a closed symbolic expression, we can use the program slice itself (i.e., the sequence of program instructions that transforms an input value into its corresponding output, according to the functionality of the program).

Given a program slice or the corresponding symbolic expression, an unknown program can be monitored. Whenever this program invokes a system call x , we extract the relevant arguments and return value. This value is then used as input to the slice or symbolic expression, computing the expected output. Later, whenever a system call y is invoked, we check its arguments. When the value of the system call argument is equal to the previously-computed, expected output, then the system has detected the data flow.

Using data flow information that is computed in the previously described fashion, we can increase the precision of matching observed system calls against the behavior graph. That is, we can make sure that a graph with a relationship $x \rightarrow y$ is matched only when we observe x and y , **and** there is a data flow between x and y that corresponds to the semantics of the malware program that is captured by this graph. As a result, we can perform more accurate detection and reduce the false positive rate.

3 System Details

In this section, we provide more details on the components of our system. In particular, we first discuss how we characterize program activity via behavior graphs. Then, we introduce our techniques to automatically ex-

tract such graphs from observing binaries. Finally, we present our approach to match the actions of an unknown binary to previously-generated behavior graphs.

3.1 Behavior Graphs: Specifying Program Activity

As a first step, we require a mechanism to describe the activity of programs. According to previous work [11], such a specification language for malicious behaviors has to satisfy three requirements: First, a specification must not constrain independent operations. The second requirement is that a specification must relate dependent operations. Third, the specification must only contain security-relevant operations.

The authors in [11] propose *malspecs* as a means to capture program behavior. A malicious specification (malspec) is a directed acyclic graph (DAG) with nodes labeled using system calls from an alphabet Σ and edges labeled using logic formulas in a logic \mathcal{L}_{dep} . Clearly, malspecs satisfy the first two requirements. That is, independent nodes (system calls) are not connected, while related operations are connected via a series of edges. The paper also mentions a function *IsTrivialComponent* that can identify and remove parts of the graph that are not security-relevant (to meet the third requirement).

For this work, we use a formalism called *behavior graphs*. Behavior graphs share similarities with malspecs. In particular, we also express program behavior as directed acyclic graphs where nodes represent system calls. However, we do not have unconstrained system call arguments, and the semantics of edges is somewhat different.

We define a system call $s \in \Sigma$ as a function that maps a set of input arguments a_1, \dots, a_n into a set of output values o_1, \dots, o_k . For each input argument of a system call a_i , the behavior graph captures where the value of this argument is derived from. For this, we use a function $f_{a_i} \in F$. Before we discuss the nature of the functions in F in more detail, we first describe where a value for a system call can be derived from. A system call value can come from three possible sources (or a mix thereof): First, it can be derived from the output argument(s) of previous system calls. Second, it can be read from the process address space (typically, the initialized data section – the `bss` segment). Third, it can be produced by the immediate argument of a machine instruction.

As mentioned previously, a function is used to capture the input to a system call argument a_i . More precisely, the function f_{a_i} for an argument a_i is defined as $f_{a_i} : x_1, x_2, \dots, x_n \rightarrow y$, where each x_i represents the output o_j of a previous system call. The values that are read from memory are part of the function body, represented by $l(addr)$. When the function is evaluated,

$l(addr)$ returns the value at memory location $addr$. This technique is needed to ensure that values that are loaded from memory (for example, keys) are not constant in the specification, but read from the process under analysis. Of course, our approach implies that the memory addresses of key data structures do not change between (polymorphic) variants of a certain malware family. In fact, this premise is confirmed by a recent observation that data structures are stable between different samples that belong to the same malware class [13]. Finally, constant values produced by instructions (through immediate operands) are implicitly encoded in the function body. Consider the case in which a system call argument a_i is the constant value 0, for example, produced by a `push $0` instruction. Here, the corresponding function is a constant function with no arguments $f_{a_i} : \rightarrow 0$. Note that a function $f \in F$ can be expressed in two different forms: As a (symbolic) formula or as an algorithm (more precisely, as a sequence of machine instructions – this representation is used in case the relation is too complex for a mathematical expression).

Whenever an input argument a_i for system call y depends on the some output o_j produced by system call x , we introduce an edge from the node that corresponds to x , to the node that corresponds to y . Thus, edges encode dependencies (i.e., temporal relationships) between system calls.

Given the previous discussion, we can define behavior graphs G more formally as: $G = (V, E, F, \delta)$, where:

- V is the set of vertices, each representing a system call $s \in \Sigma$
- E is the set of edges, $E \subseteq V \times V$
- F is the set of functions $\bigcup f : x_1, x_2, \dots, x_n \rightarrow y$, where each x_i is an output arguments o_j of system call $s \in \Sigma$
- δ , which assigns a function f_i to each system call argument a_i

Intuitively, a behavior graph encodes relationships between system calls. That is, the functions f_i for the arguments a_i of a system call s determine how these arguments depend on the outputs of previous calls, as well as program constants and memory values. Note that these functions allow one to *pre-compute* the expected arguments of a system call. Consider a behavior graph G where an input argument a of a system call s_t depends on the outputs of two previous calls s_p and s_q . Thus, there is a function f_a associated with a that has two inputs. Once we observe s_p and s_q , we can use the outputs o_p and o_q of these system calls and plug them into

f_a . At this point, we know the expected value of a , assuming that the program execution follows the semantics encoded in the behavior graph. Thus, when we observe at a later point the invocation of s_t , we can check whether its actual argument value for a matches our pre-computed value $f_a(o_p, o_q)$. If this is the case, we have high confidence that the program executes a system call whose input is related (depends on) the outputs of previous calls. This is the key idea of our proposed approach: We can identify relationships between system calls without tracking any information at the instruction-level during runtime. Instead, we rely solely on the analysis of system call arguments and the functions in the behavior graph that capture the semantics of the program.

3.2 Extracting Behavior Graphs

As mentioned in the previous section, we express program activity as behavior graphs. In this section, we describe how these behavior graphs can be automatically constructed by observing the execution of a program in a controlled environment.

Initial Behavior Graph

As a first step, an unknown malware program is executed in an extended version of Anubis [6, 7], our dynamic malware analysis environment. Anubis records all the disassembled instructions (and the system calls) that the binary under analysis executes. We call this sequence of instructions an *instruction log*. In addition, Anubis also extracts data dependencies using taint analysis. That is, the system taints (marks) each byte that is returned by a system call with a unique label. Then, we keep track of each labeled byte as the program execution progresses. This allows us to detect that the output (result) of one system call is used as an input argument for another, later system call.

While the instruction log and the taint labels provide rich information about the execution of the malware program, this information is not sufficient. Consider the case in which an instruction performs an indirect memory access. That is, the instruction *reads* a memory value from a location \mathcal{L} whose address is given in a register or another memory location. In our later analysis, we need to know which instruction was the last one to *write* to this location \mathcal{L} . Unfortunately, looking at the disassembled instruction alone, this is not possible. Thus, to make the analysis easier in subsequent steps, we also maintain a *memory log*. This log stores, for each instruction that accesses memory, the locations that this instruction reads from and writes to.

Another problem is that the previously-sketched taint tracking approach only captures data dependencies. For

example, when data is written to a file that is previously read as part of a copy operation, our system would detect such a dependency. However, it does not consider control dependencies. To see why this might be relevant, consider that the amount of data written as part of the copy operation is determined by the result of a system call that returns the size of the file that is read. The file size returned by the system call might be used in a loop that controls how often a new block of data needs to be copied. While this file size has an indirect influence on the (number of) write operation, there is no data dependency. To capture indirect dependencies, our system needs to identify the scope of code blocks that are controlled by tainted data. The start of such code blocks is identified by checking for branch operations that use tainted data as arguments. To identify the end of the scope, we leverage a technique proposed by Zhang et al. [56]. More precisely, we employ their *no preprocessing without caching* algorithm to find convergence points in the instruction log that indicate that the different paths of a conditional statement or a loop have met, indicating the end of the (dynamic) scope. Within a tainted scope, the results of all instructions are marked with the label(s) used in the branch operation, similar to the approach presented in [22].

At this point, our analysis has gathered the complete log of all executed instructions. Moreover, operands of all instructions are marked with taint labels that indicate whether these operands have data or control dependencies on the output of previous system calls. Based on this information, we can construct an initial behavior graph. To this end, every system call is mapped into a node in the graph, labeled with the name of this system call. Then, an edge is introduced from node x to y when the output of call x produces a taint label that is used in any input argument for call y .

Figure 1 depicts a part of the behavior graph of the Netsky worm. In this graph, one can see the system calls that are invoked and the dependencies between them when Netsky creates a copy of itself. The worm first obtains the name of its executable by invoking the *GetModuleFileNameA* function. Then, it opens the file of its executable by using the *NtCreateFile* call. At the same time, it creates a new file in the Windows system directory (i.e., in C:\Windows) that it calls *AVProtect9x.exe*. Obviously, the aim is to fool the user into believing that this file is benign and to improve the chances of survival of the worm. Hence, if the file is discovered by chance, a typical user will probably think that it belongs to some anti-virus software. In the last step, the worm uses the *NtCreateSection* system call to create a virtual memory block with a handle to itself and starts reading its own code and making a copy of it into the *AVProtect9x.exe* file.

In this example, the behavior graph that we generate specifically contains the string *AVProtect9x.exe*. However, obviously, a virus writer might choose to use random names when creating a new file. In this case, our behavior graph would contain the system calls that are used to create this random name. Hence, the randomization routines that are used (e.g., checking the current time and appending a constant string to it) would be a part of the behavior specification.

Figure 2 shows an excerpt of the trace that we recorded for Netsky. This is part of the input that the behavior graph is built from. On Line 1, one can see that the worm obtains the name of executable of the current process (i.e., the name of its own file). Using this name, it opens the file on Line 3 and obtains a handle to it. On Line 5, a new file called *AVProtect9x.exe* is created, where the virus will copy its code to. On Lines 8 to 10, the worm reads its own program code, copying itself into the newly created file.

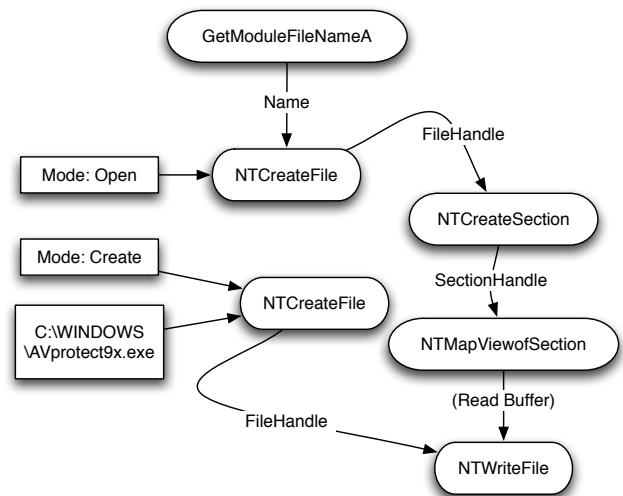


Figure 1: Partial behavior graph for Netsky.

Computing Argument Functions

In the next step, we have to compute the functions $f \in F$ that are associated with the arguments of system call nodes. That is, for each system call argument, we first have to identify the *sources* that can influence the value of this argument. Also, we need to determine how the values from the sources are manipulated to derive the argument value. For this, we make use of binary *program slicing*. Finally, we need to translate the sequence of instructions that compute the value of an argument (based on the values of the sources) into a function.

```

1  GetModuleFileNameA([out] lpFilename -> "C:\
   netsky.exe")
2  ...
3  NtCreateFile(Attr->ObjectName:"C:\netsky.exe",
   mode: open, [out] FileHandle -> A)
4  ...
5  NtCreateFile(Attr->ObjectName:"C:\WINDOWS\
   AVprotect9x.exe", mode: create, [out]
   FileHandle -> B)
6  ...
7  NtCreateSection(FileHandle: A, [out]
   SectionHandle -> C)
8  NtMapViewOfSection(SectionHandle: C,
   BaseAddress: 0x3b0000)
9  ...
10 NtWriteFile(FileHandle: B, Buffer: "MZ\90\00...
   ", Length: 16896)
11 ...

```

Figure 2: Excerpt of the observed trace for Netsky.

Program slicing. The goal of the program slicing process is to find all sources that directly or indirectly influence the value of an argument a of system call s , which is also called a *sink*. To this end, we first use the function signature for s to determine the type and the size of argument a . This allows us to determine the bytes that correspond to the sink a . Starting from these bytes, we use a standard dynamic slicing approach [2] to go backwards, looking for instructions that *define* one of these bytes. For each instruction found in this manner, we look at its operands and determine which values the instruction *uses*. For each value that is used, we locate the instruction that defines this value. This process is continued recursively. As mentioned previously, it is sometimes not sufficient to look at the instruction log alone to determine the instruction that has defined the value in a certain memory location. To handle these cases, we make use of the memory log, which helps us to find the previous write to a certain memory location.

Following def-use chains would only include instructions that are related to the sink via data dependencies. However, we also wish to include control flow dependencies into a slice. Recall from the previous subsection that our analysis computes tainted scopes (code that has a control flow dependency on a certain tainted value). Thus, when instructions are included into a slice that are within a tainted scope, the instructions that create this scope are also included, as well as the code that those instructions depend upon.

The recursive analysis chains increasingly add instructions to a slice. A chain terminates at one of two possible endpoints. One endpoint is the system call that produces a (tainted) value as output. For example, consider that we trace back the bytes that are written to a file (the argument that represents the write buffer). The analysis might determine that these bytes originate from a system call that reads the data from the network. That is, the val-

ues come from the “outside,” and we cannot go back any further. Of course, we expect that there are edges from all sources to the sink that eventually uses the values produced by the sources. Another endpoint is reached when a value is produced as an immediate operand of an instruction or read from the statically initialized data segment. In the previous example, the bytes that are written to the file need not have been read previously. Instead, they might be originating from a string embedded in the program binary, and thus, coming from “within.”

When the program slicer finishes for a system call argument a , it has marked all instructions that are involved in computing the value of a . That is, we have a subset (a slice) of the instruction log that “explains” (1) how the value for a was computed, and (2), which sources were involved. As mentioned before, these sources can be constants produced by the immediate operands of instructions, values read from memory location $addr$ (without any other instruction previously writing to this address), and the output of previous system calls.

Translating slices into functions. A program slice contains all the instructions that were involved in computing a specific value for a system call argument. However, this slice is not a program (a function) that can be directly run to compute the outputs for different inputs. A slice can (and typically does) contain a single machine instruction of the binary program more than once, often with different operands. For example, consider a loop that is executed multiple times. In this case, the instructions of the binary that make up the loop body appear multiple times in the slice. However, for our function, we would like to have code that represents the loop itself, not the unrolled version. This is because when a different input is given to the loop, it might execute a different number of times. Thus, it is important to represent the function as the actual loop code, not as an unrolled sequence of instruction.

To translate a slice into a self-contained program, we first mark all instructions in the binary that appear at least once in the slice. Note that our system handles packed binaries. That is, when a malware program is packed, we consider the instructions that it executes *after* the unpacking routine as the relevant binary code. All instructions that do not appear in the slice are replaced with no operation statements (nops). The input to this code depends on the sources of the slice. When a source is a constant, immediate operand, then this constant is directly included into the function. When the source is a read operation from a memory address $addr$ that was not previously written by the program, we replace it with a special function that reads the value at $addr$ when a program is analyzed. Finally, outputs of previous system calls are replaced with variables.

In principle, we could now run the code as a function, simply providing as input the output values that we observe from previous system calls. This would compute a result, which is the pre-computed (expected) input argument for the sink. Unfortunately, this is not that easy. The reason is that the instructions that make up the function are taken from a binary program. This binary is made up of procedures, and these procedures set up stack frames that allow them to access local variables via offsets to the base pointer (register `%ebp`) or the stack pointer (x86 register `%esp`). The problem is that operations that manipulate the base pointer or the stack pointer are often not part of the slice. As a result, they are also not part of the function code. Unfortunately, this means that local variable accesses do not behave as expected. To compensate for that, we have to go through the instruction log (and the program binary) and *fix the stack*. More precisely, we analyze the code and add appropriate instructions that manipulate the stack and, if needed, the frame pointer appropriately so that local variable accesses succeed. For this, some knowledge about compiler-specific mechanisms for handling procedures and stack frames is required. Currently, our prototype slicer is able to handle machine code generated from standard C and C++ code, as well as several human-written/optimized assembler code idioms that we encountered (for example, code that is compiled without the frame pointer).

Once the necessary code is added to fix the stack, we have a function (program) at our disposal that captures the *semantics* of that part of the program that computes a particular system call argument based on the results of previous calls. As mentioned before, this is useful, because it allows us to pre-compute the argument of a system call that we would expect to see when an unknown program exhibits behavior that conforms to our behavior graph.

Optimizing Functions

Once we have extracted a slice for a system call argument and translated it into a corresponding function (program), we could stop there. However, many functions implement a very simple behavior; they copy a value that is produced as output of a system call into the input argument of a subsequent call. For example, when a system call such as `NtOpenFile` produces an opaque handle, this handle is used as input by all subsequent system calls that operate on this file. Unfortunately, the chain of copy operations can grow quite long, involving memory accesses and stack manipulation. Thus, it would be beneficial to identify and simplify instruction sequences. Optimally, the complete sequence can be translated into a

formula that allows us to directly compute the expected output based on the formula's inputs.

To simplify functions, we make use of symbolic execution. More precisely, we assign symbolic values to the input parameters of a function and use a symbolic execution engine developed previously [23]. Once the symbolic execution of the function has finished, we obtain a symbolic expression for the output. When the symbolic execution engine does not need to perform any approximations (e.g., widening in the case of loops), then we can replace the algorithmic representation of the slice with this symbolic expression. This allows us to significantly shorten the time it takes to evaluate functions, especially those that only move values around. For complex functions, we fall back to the explicit machine code representation.

3.3 Matching Behavior Graphs

For every malware program that we analyze in our controlled environment, we automatically generate a behavior graph. These graphs can then be used for detection at the end host. More precisely, for detection, we have developed a scanner that monitors the system call invocations (and arguments) of a program under analysis. The goal of the scanner is to efficiently determine whether this program exhibits activity that matches one of the behavior graphs. If such a match occurs, the program is considered malicious, and the process is terminated. We could also imagine a system that unrolls the persistent modifications that the program has performed. For this, we could leverage previous work [45] on safe execution environments.

In the following, we discuss how our scanner matches a stream of system call invocations (received from the program under analysis) against a behavior graph. The scanner is a user-mode process that runs with administrative privileges. It is supported by a small kernel-mode driver that captures system calls and arguments of processes that should be monitored. In the current design, we assume that the malware process is running under the normal account of a user, and thus, cannot subvert the kernel driver or attack the scanner. We believe that this assumption is reasonable because, for recent versions of Windows, Microsoft has made significant effort to have users run without root privileges. Also, processes that run executables downloaded from the Internet can be automatically started in a low-integrity mode. Interestingly, we have seen malware increasingly adapting to this new landscape, and a substantial fraction can now successfully execute as a normal user.

The basic approach of our matching algorithm is the following: First, we partition the nodes of a behavior graph into a set of *active* nodes and a set of *inactive*

nodes. The set of active nodes contains those nodes that have already been matched with system call(s) in the stream. Initially, all nodes are inactive.

When a new system call s arrives, the scanner visits all inactive nodes in the behavior graph that have the correct type. That is, when a system call `NtOpenFile` is seen, we examine all inactive nodes that correspond to an `NtOpenFile` call. For each of these nodes, we check whether all its parent nodes are active. A parent node for node N is a node that has an edge to N . When we find such a node, we further have to ensure that the system call has the “right” arguments. More precisely, we have to check all functions $f_i : 1 \leq i \leq k$ associated with the k input arguments of the system call s . However, for performance reasons, we do not do this immediately. Instead, we only check the *simple functions*. Simple functions are those for which a symbolic expression exists. Most often, these functions check for the equality of handles. The checks for *complex functions*, which are functions that represent dependencies as programs, are deferred and optimistically assumed to hold.

To check whether a (simple) function f_i holds, we use the output arguments of the parent node(s) of N . More precisely, we use the appropriate values associated with the parent node(s) of N as the input to f_i . When the result of f_i matches the input argument to system call s , then we have a match. When all arguments associated with simple functions match, then node N can be activated. Moreover, once s returns, the values of its output parameters are stored with node N . This is necessary because the output of s might be needed later as input for a function that checks the arguments of N ’s child nodes.

So far, we have only checked dependencies between system calls that are captured by simple functions. As a result, we might activate a node y as the child of x , although there exists a complex dependency between these two system calls that is *not* satisfied by the actual program execution. Of course, at one point, we have to check these complex relationships (functions) as well. This point is reached when an *interesting* node in the behavior graph is activated. Interesting nodes are nodes that are (a) associated with security-relevant system calls and (b) at the “bottom” of the behavior graph. With security-relevant system calls, we refer to all calls that write to the file system, the registry, or the network. In addition, system calls that start new processes or system services are also security-relevant. A node is at the “bottom” of the behavior graph when it has no outgoing edges.

When an interesting node is activated, we go back in the behavior graph and check all complex dependencies. That is, for each active node, we check all complex functions that are associated with its arguments (in a way that is similar to the case for simple functions, as outlined previously). When all complex functions hold, the node

is marked as *confirmed*. If any of the complex functions associated with the input arguments of an active node N does not hold, our previous optimistic assumption has been invalidated. Thus, we deactivate N as well as all nodes in the subgraph rooted in N .

Intuitively, we use the concept of interesting nodes to capture the case in which a malware program has demonstrated a chain of activities that involve a series of system calls with non-trivial dependencies between them. Thus, we declare a match as soon as any interesting node has been confirmed. However, to avoid cases of overly generic behavior graphs, we only report a program as malware when the process of confirming an interesting node involves at least one complex dependency.

Since the confirmed activation of a single interesting node is enough to detect a malware sample, typically only a subset of the behavior graph of a malware sample is employed for detection. More precisely, each interesting node, together with all of its ancestor nodes and the dependencies between these nodes, can be used for detection independently. Each of these subgraphs is itself a behavior graph that describes a specific set of actions performed by a malware program (that is, a certain behavioral trait of this malware).

4 Evaluation

We claim that our system delivers effective detection with an acceptable performance overhead. In this section, we first analyze the detection capabilities of our system. Then, we examine the runtime impact of our prototype implementation. In the last section, we describe two examples of behavior graphs in more detail.

Name	Type
Allapple	Exploit-based worm
Bagle	Mass-mailing worm
Mytob	Mass-mailing worm
Agent	Trojan
Netsky	Mass-mailing worm
Mydoom	Mass-mailing worm

Table 1: Malware families used for evaluation.

4.1 Detection Effectiveness

To demonstrate that our system is effective in detecting malicious code, we first generated behavior graphs for six popular malware families. An overview of these families is provided in Table 1. These malware families were selected because they are very popular, both in our own malware data collection (which we obtained from

Name	Samples	Kaspersky variants	Our variants	Samples detected	Effectiveness
Allaple	50	2	1	50	1.00
Bagle	50	20	14	46	0.92
Mytob	50	32	12	47	0.94
Agent	50	20	2	41	0.82
Netsky	50	22	12	46	0.92
Mydoom	50	6	3	49	0.98
Total	300	102	44	279	0.93

Table 2: Training dataset.

Anubis [1]) and according to lists compiled by anti-virus vendors. Moreover, these families provide a good cross section of popular malware classes, such as mail-based worms, exploit-based worms, and a Trojan horse. Some of the families use code polymorphism to make it harder for signature-based scanners to detect them. For each malware family, we randomly selected 100 samples from our database. The selection was based on the labels produced by the Kaspersky anti-virus scanner and included different variants for each family. During the selection process, we discarded samples that, in our test environment, did not exhibit any interesting behavior. Specifically, we discarded samples that did not modify the file system, spawn new processes, or perform network communication. For the *Netsky* family, only 63 different samples were available in our dataset.

Detection capabilities. For each of our six malware families, we randomly selected 50 samples. These samples were then used for the extraction of behavior graphs. Table 2 provides some details on the training dataset. The “Kaspersky variants” column shows the number of different variants (labels) identified by the Kaspersky anti-virus scanner (these are variants such as *Netsky.k* or *Netsky.aa*). The “Our variants” column shows the number of different samples from which (different) behavior graphs had to be extracted before the training dataset was covered. Interestingly, as shown by the “Samples detected” column, it was not possible to extract behavior graphs for the entire training set. The reasons for this are twofold: First, some samples did not perform any interesting activity during behavior graph extraction (despite the fact that they did show relevant behavior during the initial selection process). Second, for some malware programs, our system was not able to extract valid behavior graphs. This is due to limitations of the current prototype that produced invalid slices (i.e., functions that simply crashed when executed).

To evaluate the detection effectiveness of our system, we used the behavior graphs extracted from the train-

ing dataset to perform detection on the remaining 263 samples (the test dataset). The results are shown in Table 3. It can be seen that some malware families, such as *Allaple* and *Mydoom*, can be detected very accurately. For others, the results appear worse. However, we have to consider that different malware variants may exhibit different behavior, so it may be unrealistic to expect that a behavior graph for one variant always matches samples belonging to another variant. This is further exacerbated by the fact that anti-virus software is not particularly good at classifying malware (a problem that has also been discussed in previous work [5]). As a result, the dataset likely contains mislabeled programs that belong to different malware families altogether. This was confirmed by manual inspection, which revealed that certain malware families (in particular, the *Agent* family) contain a large number of variants with widely varying behavior.

To confirm that different malware variants are indeed the root cause of the lower detection effectiveness, we then restricted our analysis to the 155 samples in the test dataset that belong to “known” variants. That is, we only considered those samples that belong to malware variants that are also present in the training dataset (according to Kaspersky labels). For this dataset, we obtain a detection effectiveness of 0.92. This is very similar to the result of 0.93 obtained on the training dataset. Conversely, if we restrict our analysis to the 108 samples that do *not* belong to a known variant, we obtain a detection effectiveness of only 0.23. While this value is significantly lower, it still demonstrates that our system is sometimes capable of detecting malware belonging to previously unknown variants. Together with the number of variants shown in Table 2, this indicates that our tool produces a behavior-based malware classification that is more general than that produced by an anti-virus scanner, and therefore, requires a smaller number of behavior graphs than signatures.

Name	Samples	Known variant samples	Samples detected	Effectiveness
Allaple	50	50	45	0.90
Bagle	50	26	30	0.60
Mytob	50	26	36	0.72
Agent	50	4	5	0.10
Netsky	13	5	7	0.54
Mydoom	50	44	45	0.90
Total	263	155	168	0.64

Table 3: Detection effectiveness.

False positives. In the next step, we attempted to evaluate the amount of false positives that our system would produce. For this, we installed a number of popular applications on our test machine, which runs Microsoft Windows XP and our scanner. More precisely, we used Internet Explorer, Firefox, Thunderbird, putty, and Notepad. For each of these applications, we went through a series of common use cases. For example, we surfed the web with IE and Firefox, sent a mail with Thunderbird (*including* an attachment), performed a remote ssh login with putty, and used notepad for writing and saving text. No false positives were raised in these tests. This was expected, since our models typically capture quite tightly the behavior of the individual malware families. However, if we omitted the checks for *complex functions* and assumed all complex dependencies in the behavior graph to hold, *all* of the above applications raised false positives. This shows that our tool’s ability to capture arbitrary data-flow dependencies and verify them at runtime is essential for effective detection. It also indicates that, in general, system call information alone (without considering complex relationships between their arguments) might not be sufficient to distinguish between legitimate and malicious behavior.

In addition to the Windows applications mentioned previously, we also installed a number of tools for performance measurement, as discussed in the following section. While running the performance tests, we also did not experience any false positives.

4.2 System Efficiency

As every malware scanner, our detection mechanism stands and falls with the performance degradation it causes on a running system. To evaluate the performance impact of our detection mechanism, we used 7-zip, a well-known compression utility, Microsoft Internet Explorer, and Microsoft Visual Studio. We performed the tests on a single-core, 1.8 GHz Pentium 4 running Windows XP with 1 GB of RAM.

For the first test, we used a command line option for 7-zip that makes it run a simple benchmark. This reflects the case in which an application is mostly performing CPU-bound computation. In another test, 7-zip was used to compress a folder that contains 215 MB of data (6,859 files in 808 subfolders). This test represents a more mixed workload. The third test consisted of using 7-zip to archive three copies of this same folder, performing no compression. This is a purely IO-bound workload. The next test measures the number of pages per second that could be rendered in Internet Explorer. For this test, we used a local copy of a large (1.5MB) web page [3]. For the final test, we measured the time required to compile and build our scanner tool using Microsoft Visual Studio. The source code of this tool consists of 67 files and over 17,000 lines of code. For all tests, we first ran the benchmark on the unmodified operating system (to obtain a baseline). Then, we enabled the kernel driver that logs system call parameters, but did not enable any user-mode detection processing of this output. Finally, we also enabled our malware detector with the full set of 44 behavior graphs.

The results are summarized in Table 4. As can be seen, our tool has a very low overhead (below 5%) for CPU-bound benchmarks. Also, it performs well in the I/O-bound experiment (with less than 10% overhead). The worst performance occurs in the compilation benchmark, where the system incurs an overhead of 39.8%. It may seem surprising at first that our tool performs worse in this benchmark than in the IO-bound archive benchmark. However, during compilation, the scanned application is performing almost 5,000 system calls per second, while in the archive benchmark, this value is around 700. Since the amount of computation performed in user-mode by our scanner increases with the number of system calls, compilation is a worst-case scenario for our tool. Furthermore, the more varied workload in the compile benchmark causes more complex functions to be evaluated. The 39.8% overhead of the compile benchmark can further be broken down into 12.2% for the

Test	Baseline	Driver		Scanner	
		Score	Overhead	Score	Overhead
7-zip (benchmark)	114 sec	117 sec	2.3%	118 sec	2.4%
7-zip (compress)	318 sec	328 sec	3.1%	333 sec	4.7%
7-zip (archive)	213 sec	225 sec	6.2%	231 sec	8.4%
IE - Rendering	0.41 page/s	0.39 pages/s	4.4%	0.39 page/s	4.4%
Compile	104 sec	117 sec	12.2%	146 sec	39.8%

Table 4: Performance evaluation.

kernel driver, 16.7% for the evaluation of *complex functions*, and 10.9% for the remaining user-mode processing. Note that the high cost of complex function evaluation could be reduced by improving our symbolic execution engine, so that less complex functions need to be evaluated. Furthermore, our prototype implementation spawns a new process every time that the verification of complex dependencies is triggered, causing unnecessary overhead. Nevertheless, we feel that our prototype performs well for common tasks, and the current overhead allows the system to be used on (most) end user’s hosts. Moreover, even in the worst case, the tool incurs significantly less overhead than systems that perform dynamic taint propagation (where the overhead is typically several times the baseline).

4.3 Examples of Behavior Graphs

To provide a better understanding of the type of behavior that is modeled by our system, we provide a short description of two behavior graphs extracted from variants of the Agent and Allapple malware families.

Agent.ffn.StartService. The *Agent.ffn* variant contains a resource section that stores chunks of binary data. During execution, the binary queries for one of these stored resources and processes its content with a simple, custom decryption routine. This routine uses a variant of XOR decryption with a key that changes as the decryption proceeds. In a later step, the decrypted data is used to overwrite the Windows system file `C:\WINDOWS\System32\drivers\ip6fw.sys`. Interestingly, rather than directly writing to the file, the malware opens the `\\.\C:` logical partition at the offset where the `ip6fw.sys` file is stored, and directly writes to that location. Finally, the malware restarts Windows XP’s integrated IPv6 firewall service, effectively executing the previously decrypted code.

Figure 3 shows a simplified behavior graph that captures this behavior. The graph contains nine nodes, connected through ten dependencies: six simple dependencies representing the reuse of previously ob-

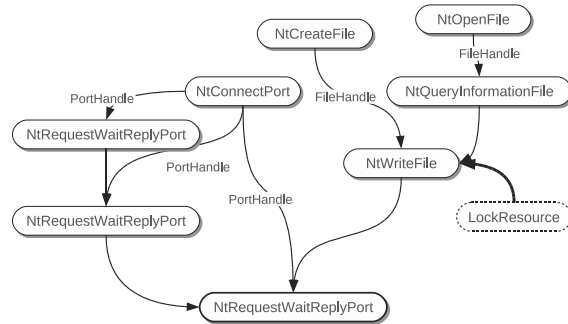


Figure 3: Behavior graph for *Agent.ffn*.

tained object handles (annotated with the parameter name), and four complex dependencies. The complex dependency that captures the previously described decryption routine is indicated by a bold arrow in Figure 3. Here, the `LockResource` function provides the body of the encrypted resource section. The `NtQueryInformationFile` call provides information about the `ip6fw.sys` file. The `\\.\C:` logical partition is opened in the `NtCreateFile` node. Finally, the `NtWriteFile` system call overwrites the firewall service program with malicious code. The check of the complex dependency is triggered by the activation of the last node (bold in the figure).

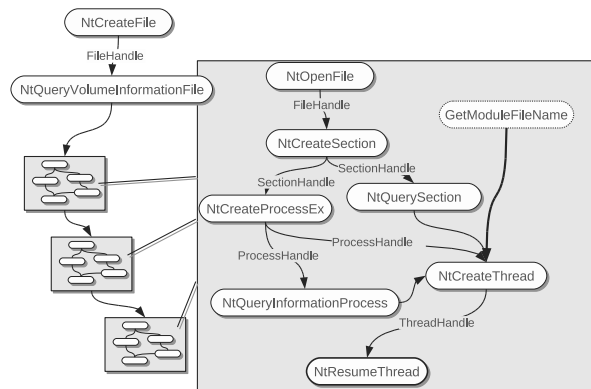


Figure 4: Behavior graph for *Allapple.b*.

Allaple.b.CreateProcess. Once started, the Allaple.b variant copies itself to the file `c:\WINDOWS\system32\urdvxc.exe`. Then, it invokes this executable various times with different command-line arguments. First, `urdvxc.exe /installservice` and `urdvxc.exe /start` are used to execute stealthily as a system service. In a second step, the malware tries to remove its traces by eliminating the original binary. This is done by calling `urdvxc.exe /uninstallservice patch:<binary>` (where `<binary>` is the name of the originally started program).

The graph shown in Figure 4 models part of this behavior. In the `NtCreateFile` node, the `urdvxc.exe` file is created. This file is then invoked three times with different arguments, resulting in three almost identical subgraphs. The box on the right-hand side of Figure 4 is an enlargement of one of these subgraphs. Here, the `NtCreateProcessEx` node represents the invocation of the `urdvxc.exe` program. The argument to the `uninstall` command (i.e., the name of the original binary) is supplied by the `GetModuleFileName` function to the `NtCreateThread` call. The last `NtResumeThread` system call triggers the verification of the complex dependencies.

5 Limitations

In this section, we discuss the limitations and possible attacks against our current system. Furthermore, we discuss possible solutions to address these limitations.

Evading signature generation. A main premise of our system is that we can observe a sample's malicious activities inside our system emulator. Furthermore, we require to find taint dependencies between data sources and the corresponding sinks. If a malware accomplishes to circumvent any of these two required steps, our system cannot generate system call signatures or find a starting point for the slicing process.

Note that our system is based on an unaccelerated version of Qemu. Since this is a system emulator (i.e., not a virtual machine), it implies that certain trivial means of detecting the virtual environment (e.g., such as Red Pill as described in [36]) are not applicable. Detecting a system emulator is an arms race against the accuracy of the emulator itself. Malware authors could also use delays, time-triggered behavior, or command and control mechanisms to try to prevent the malware from performing any malicious actions during our analysis. This is indeed the fundamental limitation of all dynamic approaches to the analysis of malicious code.

In maintaining taint label propagation, we implemented data and control dependent taint propagation and pursue a conservative approach to circumvent the loss of taint information as much as possible. Our results show that we are able to deal well with current malware. However, as soon as we observe threats in the wild targeting this feature of our system, we would need to adapt our approach.

Modifying the algorithm (input-output) behavior.

Our system's main focus lies on the detection of data input-output relations and the malicious algorithm that the malware author has created (e.g., propagation technique). As soon as a malware writer decides to implement a new algorithm (e.g., using a different propagation approach), our slices would not be usable for the this new malware type. However, note that completely modifying the malicious algorithms contained in a program requires considerable manual work as this process is difficult to automate. As a result, our system raises the bar significantly for the malware author and makes this process more costly.

6 Related Work

There is a large number of previous work that studies the behavior [34, 37, 42] or the prevalence [31, 35] of different types of malware. Moreover, there are several systems [6, 47, 54, 55] that aid an analyst in understanding the actions that a malware program performs. Furthermore, techniques have been proposed to classify malware based on its behavior using a supervised [39] or unsupervised [5, 7] learning approach. In this paper, we propose a novel technique to effectively and efficiently identify malicious code on the end host. Thus, we focus on related work in the area of malware detection.

Network detection. One line of research focuses on the development of systems that detect malicious code at the network level. Most of these systems use content-based signatures that specify tokens that are characteristic for certain malware families. These signatures can either be crafted manually [20, 33] or automatically extracted by analyzing a pool of malicious payloads [32, 41, 49]. Other approaches check for anomalous connections or for network traffic that has suspicious properties. For example, there are systems [19, 38] that detect bots based on similar connections. Other tools [50] analyze network packets for the occurrence of anomalous statistical properties. While network-based detection has the advantage that a single sensor can monitor the traffic to multiple machines, there are a number of drawbacks. First, malware has significant freedom in al-

tering network traffic, and thus, evade detection [17, 46]. Second, not all malware programs use the network to carry out their nefarious tasks. Third, even when an infected host is identified, additional action is necessary to terminate the malware program.

Static analysis. The traditional approach to detecting malware on the end host (which is implemented by anti-virus software) is based on statically scanning executables for strings or instruction sequences that are characteristic for a malware sample [46]. These strings are typically extracted from the analysis of individual programs. The problem is that such strings are typically specific to the syntactic appearance of a certain malware instance. Using code polymorphism and obfuscation, malware programs can alter their appearance while keeping their behavior (functionality) unchanged [10, 46]. As a result, they can easily evade signature-based scanners.

As a reaction to the limitations of signature-based detection, researchers have proposed a number of higher-order properties to describe executables. The hope is that such properties capture intrinsic characteristics of a malware program and thus, are more difficult to disguise. One such property is the distribution of character n-grams in a file [26, 27]. This property can help to identify embedded malicious code in other files types, for example, Word documents. Another property is the control flow graph (CFG) of an application, which was used to detect polymorphic variants of malicious code instances that all share the same CFG structure [8, 24]. More sophisticated static analysis approaches rely on code templates or specifications that capture the malicious functionality of certain malware families. Here, symbolic execution [25], model checking [21], or techniques from compiler verification [12] are applied to recognize arbitrary code fragments that implement a specific function. The power of these techniques lies in the fact that a certain functionality can always be identified, independent of the specific machine instructions that express it.

Unfortunately, static analysis for malware detection faces a number of significant problems. One problem is that current malware programs rely heavily on run-time packing and self-modifying code [46]. Thus, the instruction present in the binary on disk are typically different than those executed at runtime. While generic unpackers [40] can sometimes help to obtain the actual instructions, binary analysis of obfuscated code is still very difficult [30]. Moreover, most advanced, static analysis approaches are very slow (in the order of minutes for one sample [12]). This makes them unsuitable for detection in real-world deployment scenarios.

Dynamic analysis. Dynamic analysis techniques detect malicious code by analyzing the execution of a pro-

gram or the effects that this program has on the platform (operating system). An example of the latter category is Strider GhostBuster [52]. The tool compares the view of the system provided by a possible compromised OS to the view that is gathered when accessing the file system directly. This can detect the presence of certain types of rootkits that attempt to hide from the user by filtering the results of system calls. Another general, dynamic malware detection technique is based on the analysis of disk access patterns [16]. The basic idea is that malware activity might result in suspicious disk accesses that can be distinguished from normal program usage. The advantage of this approach is that it can be incorporated into the disk controller, and thus, is difficult to bypass. Unfortunately, it can only detect certain types of malware that scan for or modify large numbers of files.

The work that most closely relates to our own is Christodorescu et al. [11]. In [11], malware specifications (*malspecs*) are extracted by contrasting the behavior of a malware instance against a corpus of benign behaviors. Similarly to our behavior graphs, *malspecs* are DAGs where each node corresponds to a system call invocation. However, *malspecs* do not encode arbitrary data flow dependencies between system call parameters, and are therefore less specific than the behavior graphs described in this work. As discussed in Section 4, using behavior graphs for detection without verifying that complex dependencies hold would lead to an unacceptably large number of false positives.

In [22], a dynamic spyware detector system is presented that feeds browser events into Internet Explorer Browser Helper Objects (i.e., BHOs – IE plugins) and observes how the BHOs react to these browser events. An improved, tainting-based approach called Tquana is presented in [15]. In this system, memory tainting on a modified Qemu analysis environment is used to track the information that flows through a BHO. If the BHO collects sensitive data, writes this data to the disk, or sends this data over the network, the BHO is considered to be suspicious. In Panorama [55], whole-system taint analysis is performed to detect malicious code. The taint sources are typically devices such as a network card or the keyboard. In [44], bots are detected by using taint propagation to distinguish between behavior that is initiated locally and behavior that is triggered by remote commands over the network. In [29], malware is detected using a hierarchy of manually crafted behavior specifications. To obtain acceptable false positive rates, taint tracking is employed to determine whether a behavior was initiated by user input.

Although such approaches may be promising in terms of detection effectiveness, they require taint tracking on the end host to be able to perform detection. Tracking taint information across the execution of arbi-

trary, untrusted code typically requires emulation. This causes significant performance overhead, making such approaches unsuitable for deployment on end user's machines. In contrast, our system employs taint tracking when extracting a model of behavior from malicious code, but it does *not* require tainting to perform detection based on that model. Our system can, therefore, efficiently and effectively detect malware on the end user's machine.

7 Conclusion

Although a considerable amount of research effort has gone into malware analysis and detection, malicious code still remains an important threat on the Internet today. Unfortunately, the existing malware detection techniques have serious shortcomings as they are based on ineffective detection models. For example, signature-based techniques that are commonly used by anti-virus software can easily be bypassed using obfuscation or polymorphism, and system call-based approaches can often be evaded by system call reordering attacks. Furthermore, detection techniques that rely on dynamic analysis are often strong, but too slow and hence, inefficient to be used as real-time detectors on end user machines.

In this paper, we proposed a novel malware detection approach. Our approach is both *effective* and *efficient*, and thus, can be used to replace or complement traditional AV software at the end host. Our detection models cannot be easily evaded by simple obfuscation or polymorphic techniques as we try to distill the behavior of malware programs rather than their instance-specific characteristics. We generate these fine-grained models by executing the malware program in a controlled environment, monitoring and observing its interactions with the operating system. The malware detection then operates by matching the automatically-generated behavior models against the runtime behavior of unknown programs.

Acknowledgments

The authors would like to thank Christoph Karlberger for his invaluable programming effort and advice concerning the Windows kernel driver. This work has been supported by the Austrian Science Foundation (FWF) and by Secure Business Austria (SBA) under grants P-18764, P-18157, and P-18368, and by the European Commission through project FP7-ICT-216026-WOMBAT. Xiaoyong Zhou and XiaoFeng Wang were supported in part by the National Science Foundation Cyber Trust program under Grant No. CNS-0716292.

References

- [1] ANUBIS. <http://anubis.isecclab.org>, 2009.
- [2] AGRAWAL, H., AND HORGAN, J. Dynamic Program Slicing. In *Conference on Programming Language Design and Implementation (PLDI)* (1990).
- [3] B. COLLINS-SUSSMAN, B. W. FITZPATRICK AND C. M. PILATO. Version Control with Subversion. <http://svnbook.red-bean.com/en/1.5/svn-book.html>, 2008.
- [4] BAECHER, P., KOETTER, M., HOLZ, T., DORNSEIF, M., AND FREILING, F. The Nepenthes Platform: An Efficient Approach To Collect Malware. In *Recent Advances in Intrusion Detection (RAID)* (2006).
- [5] BAILEY, M., OBERHEIDE, J., ANDERSEN, J., MAO, Z., JAHANIAN, F., AND NAZARIO, J. Automated Classification and Analysis of Internet Malware. In *Symposium on Recent Advances in Intrusion Detection (RAID)* (2007).
- [6] BAYER, U., KRUEGEL, C., AND KIRDA, E. TTAalyze: A Tool for Analyzing Malware. In *Annual Conference of the European Institute for Computer Antivirus Research (EICAR)* (2006).
- [7] BAYER, U., MILANI COMPARETTI, P., HLAUSCHEK, C., KRUEGEL, C., AND KIRDA, E. Scalable, Behavior-Based Malware Clustering. In *Network and Distributed System Security Symposium (NDSS)* (2009).
- [8] BRUSCHI, D., MARTIGNONI, L., AND MONGA, M. Detecting Self-Mutating Malware Using Control Flow Graph Matching. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)* (2006).
- [9] CHRISTODORESCU, M., AND JHA, S. Static Analysis of Executables to Detect Malicious Patterns. In *Usenix Security Symposium* (2003).
- [10] CHRISTODORESCU, M., AND JHA, S. Testing Malware Detectors. In *ACM International Symposium on Software Testing and Analysis (ISSTA)* (2004).
- [11] CHRISTODORESCU, M., JHA, S., AND KRUEGEL, C. Mining Specifications of Malicious Behavior. In *European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering* (2007).
- [12] CHRISTODORESCU, M., JHA, S., SESHIA, S., SONG, D., AND BRYANT, R. Semantics-Aware Malware Detection. In *IEEE Symposium on Security and Privacy* (2005).
- [13] COZZIE, A., STRATTON, F., XUE, H., AND KING, S. Digging For Data Structures. In *Symposium on Operating Systems Design and Implementation (OSDI)* (2008).
- [14] DAGON, D., GU, G., LEE, C., AND LEE, W. A Taxonomy of Botnet Structures. In *Annual Computer Security Applications Conference (ACSAC)* (2007).
- [15] EGELE, M., KRUEGEL, C., KIRDA, E., YIN, H., AND SONG, D. Dynamic Spyware Analysis. In *Usenix Annual Technical Conference* (2007).
- [16] FELT, A., PAUL, N., EVANS, D., AND GURUMURTHI, S. Disk Level Malware Detection. In *Poster: 15th Usenix Security Symposium* (2006).
- [17] FOGLA, P., SHARIF, M., PERDISCI, R., KOLESNIKOV, O., AND LEE, W. Polymorphic Blending Attacks. In *15th Usenix Security Symposium* (2006).
- [18] FORREST, S., HOFMEYR, S., SOMAYAJI, A., AND LONGSTAFF, T. A Sense of Self for Unix Processes. In *IEEE Symposium on Security and Privacy* (1996).
- [19] GU, G., PERDISCI, R., ZHANG, J., AND LEE, W. Bot-Miner: Clustering Analysis of Network Traffic for Protocol- and Structure-Independent Botnet Detection. In *17th Usenix Security Symposium* (2008).

- [20] GU, G., PORRAS, P., YEGNESWARAN, V., FONG, M., AND LEE, W. BotHunter: Detecting Malware Infection Through IDS-Driven Dialog Correlation. In *16th Usenix Security Symposium* (2007).
- [21] KINDER, J., KATZENBEISSER, S., SCHALLHART, C., AND VEITH, H. Detecting Malicious Code by Model Checking. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)* (2005).
- [22] KIRDA, E., KRUEGEL, C., BANKS, G., VIGNA, G., AND KEMMERER, R. Behavior-based Spyware Detection. In *15th Usenix Security Symposium* (2006).
- [23] KRUEGEL, C., KIRDA, E., MUTZ, D., ROBERTSON, W., AND VIGNA, G. Automating Mimicry Attacks Using Static Binary Analysis. In *14th Usenix Security Symposium* (2005).
- [24] KRUEGEL, C., KIRDA, E., MUTZ, D., ROBERTSON, W., AND VIGNA, G. Polymorphic Worm Detection Using Structural Information of Executables. In *Symposium on Recent Advances in Intrusion Detection (RAID)* (2005).
- [25] KRUEGEL, C., ROBERTSON, W., AND VIGNA, G. Detecting Kernel-Level Rootkits Through Binary Analysis. In *Annual Computer Security Applications Conference (ACSAC)* (2004).
- [26] LI, W., STOLFO, S., STAVROU, A., ANDROULAKI, E., AND KEROMYTIS, A. A Study of Malcode-Bearing Documents. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)* (2007).
- [27] LI, W., WANG, K., STOLFO, S., AND HERZOG, B. Fileprints: Identifying File Types by N-Gram Analysis. In *IEEE Information Assurance Workshop* (2005).
- [28] LI, Z., WANG, X., LIANG, Z., AND REITER, M. AGIS: Automatic Generation of Infection Signatures. In *Conference on Dependable Systems and Networks (DSN)* (2008).
- [29] MARTIGNONI, L., STINSON, E., FREDRIKSON, M., JHA, S., AND MITCHELL, J. C. A Layered Architecture for Detecting Malicious Behaviors. In *Symposium on Recent Advances in Intrusion Detection (RAID)* (2008).
- [30] MOSER, A., KRUEGEL, C., AND KIRDA, E. Limits of Static Analysis for Malware Detection. In *23rd Annual Computer Security Applications Conference (ACSAC)* (2007).
- [31] MOSHCHUK, A., BRAGIN, T., GRIBBLE, S., AND LEVY, H. A Crawler-based Study of Spyware on the Web. In *Network and Distributed Systems Security Symposium (NDSS)* (2006).
- [32] NEWSOME, J., KARP, B., AND SONG, D. Polygraph: Automatically Generating Signatures for Polymorphic Worms. In *IEEE Symposium on Security and Privacy* (2005).
- [33] PAXSON, V. Bro: A System for Detecting Network Intruders in Real-Time. *Computer Networks* 31 (1999).
- [34] POLYCHRONAKIS, M., MAVROMMATIS, P., AND PROVOS, N. Ghost turns Zombie: Exploring the Life Cycle of Web-based Malware. In *Usenix Workshop on Large-Scale Exploits and Emergent Threats (LEET)* (2008).
- [35] PROVOS, N., MAVROMMATIS, P., RAJAB, M., AND MONROSE, F. All Your iFrames Point to Us. In *17th Usenix Security Symposium* (2008).
- [36] RAFFETSEDER, T., KRUEGEL, C., AND KIRDA, E. Detecting System Emulators. In *Information Security Conference (ISC)* (2007).
- [37] RAJAB, M., ZARFOSS, J., MONROSE, F., AND TERZIS, A. A Multifaceted Approach to Understanding the Botnet Phenomenon. In *Internet Measurement Conference (IMC)* (2006).
- [38] REITER, M., AND YEN, T. Traffic aggregation for malware detection. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)* (2008).
- [39] RIECK, K., HOLZ, T., WILLEMS, C., DUESSEL, P., AND LASKOV, P. Learning and classification of malware behavior. In *Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)* (2008).
- [40] ROYAL, P., HALPIN, M., DAGON, D., EDMONDS, R., AND LEE, W. PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware. In *Annual Computer Security Application Conference (ACSAC)* (2006).
- [41] SINGH, S., ESTAN, C., VARGHESE, G., AND SAVAGE, S. Automated Worm Fingerprinting. In *Symposium on Operating Systems Design and Implementation (OSDI)* (2004).
- [42] SMALL, S., MASON, J., MONROSE, F., PROVOS, N., AND STUBBLEFIELD, A. To Catch A Predator: A Natural Language Approach for Eliciting Malicious Payloads. In *17th Usenix Security Symposium* (2008).
- [43] SPITZNER, L. *Honeypots: Tracking Hackers*. Addison-Wesley, 2002.
- [44] STINSON, E., AND MITCHELL, J. C. Characterizing bots' remote control behavior. In *Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (2007).
- [45] SUN, W., LIANG, Z., VENKATAKRISHNAN, V., AND SEKAR, R. One-way Isolation: An Effective Approach for Realizing Safe Execution Environments. In *Network and Distributed Systems Symposium (NDSS)* (2005).
- [46] SZOR, P. *The Art of Computer Virus Research and Defense*. Addison Wesley, 2005.
- [47] VASUDEVAN, A., AND YERRABALLI, R. Cobra: Fine-grained Malware Analysis using Stealth Localized-Executions. In *IEEE Symposium on Security and Privacy* (2006).
- [48] WAGNER, D., AND DEAN, D. Intrusion Detection via Static Analysis. In *IEEE Symposium on Security and Privacy* (2001).
- [49] WANG, H., JHA, S., AND GANAPATHY, V. NetSpy: Automatic Generation of Spyware Signatures for NIDS. In *Annual Computer Security Applications Conference (ACSAC)* (2006).
- [50] WANG, K., AND STOLFO, S. Anomalous Payload-based Network Intrusion Detection. In *Symposium on Recent Advances in Intrusion Detection (RAID)* (2005).
- [51] WANG, Y., BECK, D., JIANG, X., ROUSSEV, R., VERBOWSKI, C., CHEN, S., AND KING, S. Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites That Exploit Browser Vulnerabilities. In *Network and Distributed System Security Symposium (NDSS)* (2006).
- [52] WANG, Y., BECK, D., VO, B., ROUSSEV, R., AND VERBOWSKI, C. Detecting Stealth Software with Strider Ghostbuster. In *Conference on Dependable Systems and Networks (DSN)* (2005).
- [53] WEISER, M. Program Slicing. In *International Conference on Software Engineering (ICSE)* (1981).
- [54] WILLEMS, C., HOLZ, T., AND FREILING, F. Toward Automated Dynamic Malware Analysis Using CWSandbox. *IEEE Security and Privacy* 2, 2007 (5).
- [55] YIN, H., SONG, D., EGELE, M., KRUEGEL, C., AND KIRDA, E. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *ACM Conference on Computer and Communication Security (CCS)* (2007).
- [56] ZHANG, X., GUPTA, R., AND ZHANG, Y. Precise dynamic slicing algorithms. In *International Conference on Software Engineering (ICSE)* (2003).

Protecting Confidential Data on Personal Computers with Storage Capsules

Kevin Borders, Eric Vander Weele, Billy Lau, and Atul Prakash

University of Michigan

Ann Arbor, MI, 48109

{kborders, ericvw, billylau, aprakash}@umich.edu

Abstract

Protecting confidential information is a major concern for organizations and individuals alike, who stand to suffer huge losses if private data falls into the wrong hands. One of the primary threats to confidentiality is malicious software on personal computers, which is estimated to already reside on 100 to 150 million machines. Current security controls, such as firewalls, anti-virus software, and intrusion detection systems, are inadequate at preventing malware infection. This paper introduces Storage Capsules, a new approach for protecting confidential files on a personal computer. Storage Capsules are encrypted file containers that allow a compromised machine to securely view and edit sensitive files without malware being able to steal confidential data. The system achieves this goal by taking a checkpoint of the current system state and disabling device output before allowing access a Storage Capsule. Writes to the Storage Capsule are then sent to a trusted module. When the user is done editing files in the Storage Capsule, the system is restored to its original state and device output resumes normally. Finally, the trusted module declassifies the Storage Capsule by re-encrypting its contents, and exports it for storage in a low-integrity environment. This work presents the design, implementation, and evaluation of Storage Capsules, with a focus on exploring covert channels.

1. Introduction

Traditional methods for protecting confidential information rely on upholding system integrity. If a computer is safe from hackers and malicious software (malware), then so is its data. Ensuring integrity in today's interconnected world, however, is exceedingly difficult. Trusted computing platforms such as Terra [8] and trusted boot [26] try to provide this integrity by verifying software. Unfortunately, these platforms are rarely deployed in practice and most software continues to be unverified. More widely-applicable security tools, such as firewalls, intrusion detection systems, and anti-virus software, have been unable to combat malware, with 100 to 150 million infected machines running on the Internet today according to a recent estimate [34]. Security mechanisms for personal computers simply cannot rely on keeping high integrity. Storage Capsules address the need for access to confidential data from compromised personal computers.

There are some existing solutions for preserving confidentiality that do not rely on high integrity. One example is mandatory access control (MAC), which is used by Security-Enhanced Linux [23]. MAC can control the flow of sensitive data with policies that prevent entities that read confidential information from communicating over the network. This policy set achieves the goal of preventing leaks in the presence of malware. However, defining correct policies can be difficult, and they would prevent most useful applications from running properly. For example, documents saved by a word

processor that has ever read secret data could not be sent as e-mail attachments. Another embodiment of the same principle can be seen in an "air gap" separated network where computers are physically disconnected from the outside world. Unplugging a compromised computer from the Internet will stop it from leaking information, but doing so greatly limits its utility. Both mandatory access control with strict outbound flow policies and air gap networks are rarely used outside of protecting classified information due to their severe impact on usability.

This paper introduces Storage Capsules, a new mechanism for protecting sensitive information on a local computer. The goal of Storage Capsules is to deliver the same level of security as a mandatory access control system for standard applications running on a commodity operating system. Storage Capsules meet this requirement by enforcing policies at a system-wide level using virtual machines. The user's system can also downgrade from high-secrecy to low-secrecy by reverting to a prior state using virtual machine snapshots. Finally, the system can obtain updated Storage Capsules from a declassification component after returning to low secrecy.

Storage Capsules are analogous to encrypted file containers from the user's perspective. When the user opens a Storage Capsule, a snapshot is taken of the current system state and device output is disabled. At this point, the system is considered to be in *secure mode*. When the user is finished editing files in a Storage Capsule, the system is reverted to its original state – dis-

carding all changes except those made to the Storage Capsule – and device output is re-enabled. The storage capsule is finally re-encrypted by a trusted component.

Storage Capsules guarantee protection against a compromised operating system or applications. Sensitive files are safe when they are encrypted *and* when being accessed by the user in plain text. The Capsule system prevents the OS from leaking information by erasing its entire state after it sees sensitive data. It also stops covert communication by fixing the Storage Capsule size and completely re-encrypting the data every time it is accessed by the OS. Our threat model assumes that the primary operating system can do anything at all to undermine the system. The threat model also assumes that the user, hardware, the virtual machine monitor (VMM), and an isolated secure virtual machine are trustworthy. The Capsule system protects against covert channels in the primary OS and Storage Capsules, as well as many (though not all) covert channels at lower layers (disk, CPU, etc.). One of the contributions of this paper is identifying and suggesting mitigation strategies for numerous covert channels that could potentially leak data from a high-secrecy VM to a low-secrecy VM that runs after it has terminated.

We evaluated the impact that Storage Capsules have on the user's workflow by measuring the latency of security level transitions and system performance during secure mode. We found that for a primary operating system with 512 MB of RAM, transitions to secure mode took about 4.5 seconds, while transitions out of secure mode took approximately 20 seconds. We also compared the performance of the Apache build benchmark in secure mode to that of a native machine, a plain virtual machine, and a virtual machine running an encryption utility. Overall, Storage Capsules added 38% overhead compared to a native machine, and only 5% compared to a VM with encryption software. The common workload for a Storage Capsule is expected to be much lighter than an Apache build. In many cases, it will add only a negligible overhead.

The main contribution of this work is a system that allows safe access to sensitive files from a normal operating system with standard applications. The Capsule system is able to switch modes within one OS rather than requiring separate operating systems or processes for different modes. This paper also makes contributions in the understanding of covert channels in such a system. In particular, it looks at how virtualization technology can create new covert channels and how previously explored covert channels behave differently when the threat model is a low-security virtual machine running after a high-security virtual machine.

It is important to keep in mind that Storage Capsules do not protect integrity. There are a number of attacks that they cannot prevent. If malicious software stops the user from ever entering secure mode by crashing, then the user might be coerced into accessing sensitive files without Storage Capsules. Furthermore, malware can manipulate data to present false information that tricks the user into doing something erroneously, such as placing a stock transaction. These attacks are beyond the scope of this paper.

The remainder of this paper is laid out as follows. Section 2 discusses related work. Section 3 gives an overview of the usage model, the threat model, and design alternatives. Section 4 outlines the system architecture. Section 5 describes the operation of Storage Capsules. Section 6 examines the effect of covert channels on Storage Capsules. Section 7 presents evaluation results. Finally, section 8 concludes and discusses future work.

2. Related Work

The Terra system [8] provides multiple security levels for virtual machines using trusted computing technology. Terra verifies each system component at startup using a trusted platform model (TPM) [29], similar to trusted boot [26]. However, Terra allows unverified code to run in low-security virtual machines. One could imagine a configuration of Terra in which the user's primary OS runs inside of a low-integrity machine, just like in the Capsule system. The user could have a separate secure VM for decrypting, editing, and encrypting files. Assuming that the secure VM always has high integrity, this approach would provide comparable security and usability benefits to Storage Capsules. However, Terra only ensures a secure VM's integrity at startup; it does not protect running software from exploitation. If this secure VM ever loads an encrypted file from an untrusted location, it is exposed to attack. All sources of sensitive data (e-mail contacts, web servers, etc.) would have to be verified and added to the trusted computing base (TCB), bloating its size and impacting both management overhead and security. Furthermore, the user would be unable to safely include data from untrusted sources, such as the internet, in sensitive files. The Capsule system imposes no such headaches; it can include low-integrity data in protected files, and only requires trust in local system components to guarantee confidentiality.

There has been extensive research on controlling the flow of sensitive information inside of a computer. Intra-process flow control techniques aim to verify that individual applications do not inadvertently leak confidential data [6, 22]. However, this does not stop mali-

cious software that has compromised a computer from stealing data at the operating system or file system level. Another approach for controlling information flow is at the process level with a mandatory access control (MAC) system like SELinux [23]. MAC involves enforcing access control policies on high-level objects (typically files, processes, etc.). However, defining correct policies can be quite difficult [15] even for a fixed set of applications. MAC would have a hard time protecting personal computers that download and install programs from the internet. Very few computers use mandatory access control currently, and it is not supported by Microsoft Windows, a popular operating system for personal computers. Storage Capsules employ a similar approach to MAC systems, but do so at a higher level of granularity (system-wide) using virtual machine technology. This allows Storage Capsules to provide more practical security for commodity operating systems without requiring modification.

There are a number of security products available for encrypting and protecting files on a local computer, including compression utilities [25, 35] and full disk encryption software [1, 7, 20, 31]. The goal of file encryption is to facilitate file transmission over an untrusted medium (e.g., an e-mail attachment), or protect against adversarial access to the storage device (e.g., a lost or stolen laptop). File encryption software does safeguard sensitive information while it is decrypted on the end host. Malicious software that has control of the end host can steal confidential data or encryption keys. Capsule also uses file encryption to allow storage in an untrusted location, but it maintains confidentiality while sensitive data is decrypted on the end host.

Storage Capsules rely on the virtual machine monitor as part of the trusted computing base. VMMs are commonly accepted as less complex and more secure than standard operating systems, with the Xen VMM having under 50,000 lines of code [36], compared to 5.7 million lines in the Linux 2.6 kernel [5]. These numbers are reinforced by actual vulnerability reports, with Xen 3.x only having 9 reports up to January 2009 [27], and the Linux 2.6.x kernel having 165 reports [28] in that same time period. VMMs are not invulnerable, but they have proven to be more robust than standard kernels.

Virtualization technology has many useful properties and features that make it a well-suited platform for Storage Capsules. Despite these advantages, Garfinkel et al. warn that virtualization has some shortcomings, especially when it comes to security [9]. Most importantly, have many branches and saved states makes patching and configuration much more difficult. A user might load an old snapshot that is vulnerable to infec-

tion by an Internet worm. The Capsule system does not suffer from these limitations because it is designed to have one primary VM with a fairly straight execution path. Transitions too and from secure mode are short-lived, and should have a minimal impact on patching and management tasks.

3. Overview

3.1 Storage Capsules from a User's Perspective

From the user's perspective, Storage Capsules are analogous to encrypted file containers provided by a program like TrueCrypt [31]. Basing the Capsule system off of an existing and popular program's usage model makes it easier to gain acceptance. The primary difference between Storage Capsules and traditional encryption software is that the system enters a secure mode before opening the Storage Capsule's contents. In this secure mode, network output is disabled and any changes that the user makes outside of the Storage Capsule will be lost. The user may still edit the Storage Capsule contents with his or her standard applications. When the user closes the Storage Capsule and exits secure mode, the system reverts to the state it was in before accessing sensitive data.

One motivating example for Storage Capsules is providing a secure journal. A person, call him Bob, may want to write a diary in which he expresses controversial political beliefs. Bob might regularly write in this journal, possibly pasting in news stories or contributions from others on the internet. Being a diligent user, Bob might store this document in an encrypted file container. Unfortunately, Bob is still completely vulnerable to spyware when he enters the decryption password and edits the document. Storage Capsules support the same usage model as normal encrypted file containers, but also deliver protection against spyware while the user is accessing sensitive data.

Storage Capsules have some limitations compared to encrypted file containers. These limitations are necessary to gain additional security. First, changes that the user makes outside of the encrypted Storage Capsule while it is open will not persist. This benefits security and privacy by eliminating all traces of activity while the container was open. Storage Capsules guarantee that the OS does not inadvertently hold information about sensitive files as described by Czeskis et al. for the case of TrueCrypt [4]. Unfortunately, any work from computational or network processes that may be running in the background will be lost. One way to remove this limitation would be to fork the primary virtual machine and

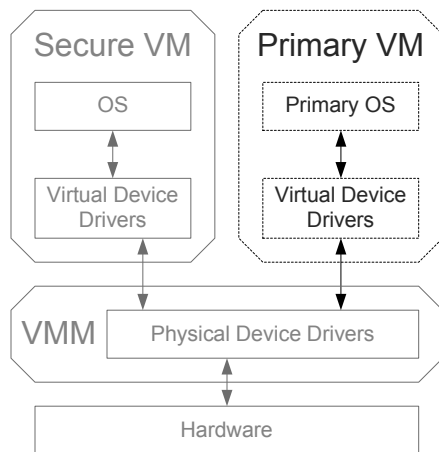


Figure 1. In the Storage Capsule architecture, the user's primary operating system runs in a virtual machine. The secure VM handles encryption and declassification. The dotted black line surrounding the primary VM indicates that it is *not* trusted. The other system components are trusted.

allow a copy of it to run in the background. Allowing low- and high-secrecy VMs to run at the same time, however, reduces security by opening up the door for a variety of covert channels.

3.2 Threat Model

Storage Capsules are designed to allow a compromised operating system to safely edit confidential information. However, some trusted components are necessary to provide security. Figure 1 shows the architecture of the Capsule system, with trusted components having solid lines and untrusted components having dotted lines. The user's primary operating system runs inside of a *primary VM*. Neither the applications, the drivers, nor the operating system are trusted in the primary VM; it can behave in any arbitrary manner. A *virtual machine monitor (VMM)* runs beneath the primary VM, and is responsible for mediating access to physical devices. The VMM is considered part of the trusted computing base (TCB). The Capsule system also relies on a *Secure VM* to save changes and re-encrypt Storage Capsules. This secure VM has only a minimal set of applications to service Storage Capsule requests, and has all other services blocked off with a firewall. The secure VM is also part of the TCB.

The user is also considered trustworthy in his or her intent. Presumably, the user has a password to decrypt each Storage Capsule and could do so using rogue software without going into secure mode and leak sensitive data. The user does not require full access to any trusted components, however. The main user interface is the

primary VM, and the user should only interact with the Secure VM or VMM briefly using a limited UI. This prevents the user from inadvertently compromising a trusted component with bad input.

The threat model assumes that malicious software may try to communicate covertly within the primary VM. Storage Capsules are designed to prevent a compromised primary OS from saving data anywhere that will persist through a snapshot restoration. However, Storage Capsules do not guarantee that a malicious primary VM cannot store data somewhere in a trusted component, such as hardware or the VMM, in such a way that it can recover information after leaving secure mode. We discuss several of these covert channels in more depth later in the paper.

3.3 Designs that do not Satisfy Storage Capsule Goals

The first system design that would not meet the security goals laid out in our threat model is conventional file encryption software [1, 7, 20, 31]. Any information stored in an encrypted file would be safe from malicious software, or even a compromised operating system, while it is encrypted. However, as soon as the user decrypts a file, the operating system can do whatever it wants with the decrypted data.

Another design that would not meet the goals of Storage Capsules is the NetTop architecture [19]. With NetTop, a user has virtual machines with multiple security levels. One is for accessing high-secrecy information, and another for low-secrecy information, which may be connected to the internet. Depending on how policies are defined, NetTop either suffers from usability limitations or would have security problems. First assume that the high-secrecy VM must be able to read data from the low-secrecy VM to load files from external locations that are not part of the trusted computing base. Now, if the high-secrecy VM is prevented from writing anything back to the low-secrecy VM, then confidentiality is maintained. However, this prevents the user from making changes to a sensitive document, encrypting it, then sending it back out over a low-secrecy medium. This effectively makes everything read-only from the high-secrecy VM to the low-secrecy VM. The other alternative – letting the high-secrecy VM encrypt and declassify data – opens up a major security hole. Data that comes from the low-secrecy VM also might be malicious in nature. If the high-secrecy VM reads that information, its integrity – and the integrity of its encryption operations – may be compromised.

4. System Architecture

The Capsule system has two primary modes of operation: *normal mode* and *secure mode*. In normal mode, the computer behaves the same as it would without the Capsule system. The primary operating system has access to all devices and can communicate freely over the network. In secure mode, the primary OS is blocked from sending output to the external network or to devices that can store data. Furthermore, the primary operating system's state is saved prior to entering secure mode, and then restored when transitioning back to normal mode. This prevents malicious software running on the primary OS from leaking data from secure mode to normal mode.

The Capsule system utilizes virtual machine technology to isolate the primary OS in secure mode. Virtual machines also make it easy to save and restore system state when transitioning to or from secure mode. Figure 1 illustrates the architecture of the Capsule system. The first virtual machine, labeled Primary VM, contains the primary operating system. This VM is the equivalent of the user's original computer. It contains all of the user's applications, settings, and documents. This virtual machine may be infected with malicious software and is not considered trustworthy. The other virtual machine, labeled Secure VM, is responsible for managing access to Storage Capsules. The secure VM is trusted. The final component of the Capsule system shown in Figure 1 is the Virtual Machine Monitor (VMM). The VMM is responsible for translating each virtual device I/O request into a physical device request, and for governing virtual networks. As such, it can also block device I/O from virtual machines. The VMM has the power to start, stop, save, and restore entire virtual machines. Because it has full control of the computer, the VMM is part of the trusted computing base.

The Capsule system adds three components to the above architecture to facilitate secure access to Storage Capsules. The first is the *Capsule VMM module*, which runs as service inside of the VMM. The Capsule VMM module performs the following basic functions:

- Saves and restores snapshots of the primary VM
- Enables and disables device access by the primary VM
- Catches key escape sequences from the user
- Switches the UI between the primary VM and the secure VM

The Capsule VMM module executes operations as requested by the second component, the *Capsule server*, which runs inside of the secure VM. The Capsule server manages transitions between normal mode and secure

mode. During secure mode, it also acts as a disk server, handling block-level read and write requests from the *Capsule viewer*, which runs in the primary VM. The Capsule server has dedicated interfaces for communicating with the Capsule viewer and with the Capsule VMM module. These interfaces are attached to separate virtual networks so that the viewer and VMM module cannot impersonate or communicate directly with each other.

The third component, the Capsule viewer, is an application that accesses Storage Capsules on the primary VM. When the user first loads or creates a new Storage Capsule, the viewer will import the file by sending it to the Capsule server. The user can subsequently open the Storage Capsule, at which point the viewer will ask the Capsule server to transition the system to secure mode. During secure mode, the viewer presents the contents of the Storage Capsule to the user as a new mounted partition. Block-level read and write requests made by the file system are forwarded by the viewer to the Capsule server, which encrypts and saves changes to the Storage Capsule. Finally, the Capsule viewer can retrieve the encrypted Storage Capsule by requesting an export from the Capsule server. The Capsule viewer is not trusted and may cause a denial-of-service at any time. However, the Capsule system is designed to prevent even a compromised viewer from leaking data from secure mode to normal mode.

5. Storage Capsule Operation

5.1 Storage Capsule File Format

A Storage Capsule is actually an encrypted partition that is mounted during secure mode. The Storage Capsule model is based on TrueCrypt [31] – a popular encrypted storage program. Like TrueCrypt, each new Storage Capsule is created with a fixed size. Storage Capsules employ XTS-AES – the same encryption scheme as TrueCrypt – which is the IEEE standard for data encryption [13]. In our implementation, the encryption key for each file is created by taking the SHA-512 hash of a user-supplied password. In a production system, it would be beneficial to employ other methods, such as hashing the password many times and adding a salt, to make attacks more difficult. The key could also come from a biometric reader (fingerprint reader, retina scanner, etc.), or be stored on a key storage device like a smart card. Storage Capsules operation does not depend on a particular key source.

With XTS-AES, a different tweak value is used during encryption for each data unit. A data unit can be one or more AES blocks. The Storage Capsule implementation

uses a single AES block for each data unit. In accordance with the IEEE 1619 standard [13], Storage Capsules use a random 128-bit starting tweak value that is incremented for each data unit. This starting tweak value is needed for decryption, so it is stored at the beginning of the file. Because knowledge of the tweak value does not weaken the encryption [18], it is stored in the clear.

5.2 Creating and Importing a Storage Capsule

The first step in securing data is creating a new Storage Capsule. The following tasks take place during the creation process:

1. The Capsule viewer solicits a Storage Capsule file name and size from the user.
2. The viewer makes a request to the Capsule server on the secure VM to create a new Storage Capsule.
3. The viewer asks the user to enter the secure key escape sequence that will be caught by a keyboard filter driver in the VMM. This deters spoofing by a compromised primary VM.
4. After receiving the escape sequence, the VMM module will give the secure VM control of the user interface.
 - a. If the escape sequence is received unexpectedly (i.e. when the VMM has not received a request to wait for an escape sequence from the Capsule server), then the VMM module will still give control of the UI to the secure VM, but the secure VM will display a warning message saying that the user is *not* at a secure password entry page.
5. The Capsule server will ask the user to select a password, choose a random starting tweak value for encryption, and then format the encapsulated partition.
6. The Capsule server asks the VMM module to switch UI focus back to the primary VM.

After the creation process is complete, the Capsule server will send the viewer a file ID that it can store locally to link to the Storage Capsule on the server.

Loading a Storage Capsule from an external location requires fewer steps than creating a new Storage Capsule. If the viewer opens a Storage Capsule file that has been created elsewhere, it imports the file by sending it to the Capsule server. In exchange, the Capsule server sends the viewer a file ID that it can use as a link to the newly imported Storage Capsule. After a Storage Capsule has been loaded, the link on the primary VM looks

the same regardless of whether the Capsule was created locally or imported from an external location.

5.3 Opening a Storage Capsule in Secure Mode

At this point, one or more Storage Capsules reside on the Capsule server, and have links to them on the primary VM. When the user opens a link with the Capsule viewer, it will begin the transition to secure mode, which consists of the following steps:

1. The Capsule viewer sends the Capsule server a message saying that the user wants to open a Storage Capsule, which includes the file ID from the link in the primary VM.
2. The Capsule viewer asks the user to enter the escape sequence that will be caught by the VMM module.
3. The VMM module receives the escape sequence and switches the UI focus to the secure VM. This prevents malware on the primary VM from spoofing a transition and stealing the file password.
 - a. If the escape sequence is received unexpectedly, the secure VM still receives UI focus, but displays a warning message stating the system is *not* in secure mode.
4. The VMM module begins saving a snapshot of the primary VM in the background. Execution continues, but memory and disk data is copied to the snapshot on write.
5. The VMM module disables network and other device output.
6. The Capsule server asks the user to enter the file password.
7. The VMM module returns UI focus to the primary VM.
8. The Capsule server tells the viewer that the transition is complete and begins servicing disk I/O requests to the Storage Capsule.
9. The Capsule viewer mounts a local partition that uses the Capsule server for back-end disk block storage.

The above process ensures that the primary VM gains access to the Storage Capsule contents only after its initial state has been saved and the VMM has blocked device output. The exact set of devices blocked during secure mode is discussed more in the section on covert channels.

Depending on the source of the Storage Capsule encryption key, step 6 could be eliminated entirely. If the key was obtained from a smart card or other device, then the primary VM would retain UI focus throughout the entire transition, except in the case of an unexpected

escape sequence from the user. In this case, the secure VM must always take over the screen and warn the user that he or she is not in secure mode.

5.4 Storage Capsule Access in Secure Mode

When the Capsule system is running in secure mode, all reads and writes to the Storage Capsule are sent to the Capsule server. The server will encrypt and decrypt the data for each request as it is received, without performing any caching itself. The Capsule server instead relies on the caches within the primary VM and its own operating system to minimize unnecessary encryption and disk I/O. The disk cache in the primary VM sits above the driver that sends requests through to the Capsule server. On the secure VM, disk read and write requests from the Capsule server go through the local file system cache before they are sent to the disk. Later, we measure Storage Capsule disk performance during secure mode and demonstrate that is comparable to current encryption and virtualization software.

During secure mode, the VMM stores all writes to the primary VM's virtual disk in a file. This file contains differences with respect to the disk state at the time of the last snapshot operation (during the transition to secure mode). In the Capsule system, this difference file is stored on a partition that resides in main memory, commonly referred to as a RAM disk. Our implementation uses the ImDisk Virtual Disk Driver [14]. Storing the files on a RAM disk prevents potentially sensitive data that the primary VM writes to its virtual disk from reaching the physical disk. Although malicious access to the physical disk is not the focus of this paper's threat model, allowing confidential data to touch the disk increases the overall risk of leakage and the cost of hardware destruction.

The main downside to storing virtual disk modifications in memory is that the system requires more memory in secure mode. However, the size of primary disk modifications in secure mode should be small under normal circumstances because they are all temporary. Under normal circumstances, the user has no reason to store big files in a location where they will soon be deleted. If the change buffer does fill up due to a denial-of-service by malware or non-standard usage, then writes to the main virtual disk will fail until the system reverts to normal mode. If there turns out to be a legitimate need for large amounts of temporary storage in secure mode, then the change buffer could be stored on the physical disk and the VMM could use an encrypted file system for added protection.

5.5 Reverting to Normal Mode

Transitioning the Capsule system from secure mode back to normal mode is the most security-sensitive operation. Care must be taken to prevent leaks from secure mode back to normal mode insofar as practical for the desired level of security. A full discussion of the channels through which information might flow during this transition and countermeasures can be found in the next section. The Capsule system begins reverting to normal mode when the user enters a key escape sequence. Here, the escape sequence is not to prevent spoofing, but instead to reduce the primary VM's ability to leak data through a timing channel. After the user hits the escape sequence, the following steps take place:

1. The VMM module notifies the Capsule server of the pending transition, which in turn notifies the Capsule viewer.
2. The Capsule server waits up to 30 seconds for the primary VM to flush disk writes to the Storage Capsule. In our experiments, flushing always took less than one second, but uncommon workloads could make it take longer. We chose 30 seconds because it is the default maximum write-back delay for linux.
3. The secure VM reboots in order to flush any state that was affected by the primary VM. (This blocks some covert channels that are discussed in the next section.)
4. The VMM module halts the primary VM, and then reverts its state to the snapshot taken before entering secure mode and resumes execution.
5. The VMM module re-enables network and other device output for the primary VM.

After the Capsule system has reverted to normal mode, all changes that were made in the primary VM during secure mode, except those to the Storage Capsule, are lost. Also, when the Capsule viewer resumes executing in normal mode, it queries the Capsule to see what mode it is in (if the connection fails due to the reboot, normal mode is assumed). This is a similar mechanism to the return value from a fork operation. Without it, the Capsule viewer cannot tell whether secure mode is just beginning or the system has just reverted to normal mode, because both modes start from the same state.

5.6 Exporting Storage Capsules

After modifying a storage capsule, the user will probably want to back it up or transfer it to another person or computer at some point. Storage Capsules support this use case by providing an export operation. The Capsule viewer may request an export from the Capsule server at any time during normal mode. When the Capsule server

exports an encrypted Storage Capsule back to the primary VM, it is essential that malicious software can glean no information from the difference between the Storage Capsule at export compared to its contents at import. This should be the case even if malware has full control of the primary VM during secure mode and can manipulate the Storage Capsule contents in a chosen-plaintext attack.

For the Storage Capsule encryption scheme to be secure, the difference between the exported cipher-text and the original imported cipher-text must appear completely random. If the primary VM can change specific parts of the exported Storage Capsule, then it could leak data from secure mode. To combat this attack, the Capsule server re-encrypts the entire Storage Capsule using a new random 128-bit starting tweak value before each export. There is a small chance of two exports colliding. For any two Storage Capsules, each of size 2 GB (2^{27} encryption blocks), the chance of random 128-bit tweak values partially colliding would be approximately 1 in $2 * 2^{27} / 2^{128}$ or 1 in 2^{100} . Because of the birthday paradox, however, there will be a reasonable chance of a collision between a pair of exports after only 2^{50} exports. This number decreases further with the size of Storage Capsules. Running that many exports would still take an extremely long time (36 million years running 1 export / second). We believe that such an attack is unlikely to be an issue in reality, but could be mitigated if future tweaked encryption schemes support 256-bit tweak values.

5.7 Key Escape Sequences

During all Capsule operations, the primary VM and the Capsule viewer are not trusted. Some steps in the Capsule system's operation involve the viewer asking the user to enter a key escape sequence. If the primary VM becomes compromised, however, it could just skip asking the user to enter escape sequences and display a spoofed UI that looks like what would show up if the user did hit the escape sequence. This attack would steal the file decryption password while the system is still in normal mode. The defense against this attack is that the user should be accustomed to entering the escape sequence and therefore hit it anyway or notice anomalous behavior.

It is unclear how susceptible real users would be to spoofing attack that omits asking for an escape sequence. The success of such an attack is likely to depend on user education. Formally evaluating the usability of escape sequences in the Capsule system is future work. Another design alternative that may help if spoofing attacks are found to be a problem is reserving a se-

cure area on the display. This area would always tell the user whether the system is in secure mode or the secure VM has control of the UI.

6. Covert Channel Analysis

The Storage Capsule system is designed to prevent any direct flow of information from secure mode to normal mode. However, there are a number of covert channels through which information may be able to persist during the transition from secure to normal mode. This section tries to answer the following questions about covert channels in the Capsule system as best as possible:

- Where can the primary virtual machine store information that it can retrieve after reverting to normal mode?
- What defenses might fully or partially mitigate these covert information channels?

We do not claim to expose all covert channels here, but list many channels that we have uncovered during our research. Likewise, the proposed mitigation strategies are not necessarily optimal, but represent possible approaches for reducing the bandwidth of covert channels. Measuring the maximum bandwidth of each covert channel requires extensive analysis and is beyond the scope of this paper. There has been a great deal of research on measuring the bandwidth of covert channels [2, 16, 21, 24, 30, 33], which could be applied to calculate the severity of covert channels in the Capsule system in future work.

The covert channels discussed in this section can be divided into five categories:

1. Primary OS and Capsule – Specific to Storage Capsule design
2. External Devices – Includes floppy, CD-ROM, USB, SCSI, etc.
3. External Network – Changes during secure mode that affect responsiveness to external connections
4. VMM – Arising from virtual machine monitor implementation, includes memory mapping and virtual devices
5. Core Hardware – Includes CPU and disk drives.

The Capsule system prevents most covert channels in the first three categories. It can use the VMM to mediate the primary virtual machine's device access and completely erase the primary VM's state when reverting to normal mode. The Capsule system also works to prevent timing channels when switching between modes of operation, and does respond to external network traffic while in secure mode.

Storage Capsules do not necessarily protect against covert channels in the last two categories. There has been a lot of work on identifying, measuring, and mitigating covert channels in core hardware for traditional MLS systems [16, 17, 21, 30]. Similar methods for measuring and mitigating those core channels could be applied to Storage Capsules. Covert channels arising from virtualization technology have not received much attention. This research hopes to highlight some of the key mechanisms in a VMM that can facilitate covert communication. The remainder of this section explores covert channels in each of these categories, including mitigation strategies and their required trade-offs.

6.1 Primary OS and Storage Capsule Channels

In addition to covert channels that one might find in other systems, the design of Storage Capsules creates opportunities for covert communication. This section focuses on the interactions between the primary VM and the secure VM. The primary VM should not be able to affect the secure VM's state during secure mode in a way that it can detect during normal mode. Here we examine three attack vectors that a compromised primary VM might use to steal data, and corresponding counter-measures.

Transition Timing

When the VMM restores the primary virtual machine from a snapshot, the restored state contains the exact same information as prior to entering secure mode. Following the restoration, the primary VM can obtain additional information by fetching the system time, which tells it when the system returned to normal mode. If the primary VM can manipulate the transition timing from secure mode, then it can leak data.

The Capsule system prevents direct leaks through transition timing by requiring a secure key escape sequence to switch back to normal mode. However, the primary VM may still be able to affect transition timing depending on how long the Capsule server waits for the primary VM to flush disk writes to the Storage Capsule upon leaving secure mode. One option that would preclude manipulation of transition timing would be for the Capsule server to wait a fixed time-period for disk writes to complete. This would be the most secure approach. Unfortunately, this strategy would either impact usability or threaten correctness. If the server were to stall 30 seconds – the default write-back delay for Linux – then the transition time would be unnecessarily long in most cases. On the other hand, if it only waited a few seconds, then some delayed writes might fail and corrupt the Storage Capsule.

For our implementation of the Capsule system, we chose a transition timing policy that sacrifices a few bits of potential information leakage per transition, but is more usable. The Capsule server will wait for the primary VM to flush Storage Capsule writes for up to 30 seconds. If the primary VM finishes the disk flush in less than 30 seconds, the server will immediately proceed with restoration. This opens up the possibility of the primary VM stalling a precise length of time after flushing writes so that it can read back timing information after returning to normal mode. The maximum bandwidth of this timing channel is somewhat limited, however, because the snapshot restoration adds noise to the transition time. Depending on how much memory the primary VM has, the amount of time between exiting secure mode and resuming in normal mode can vary by several seconds. By notifying the server at a specific offset from the most recent half-minute, the primary VM will only be able to communicate a few bits of data. One possible counter-measure to this type of timing channel would be to deliberately skew the clock during secure mode so the primary VM cannot accurately judge system time.

Storage Capsule Contents

The ideal place for a compromised primary VM to store data would be in the Storage Capsule itself. It is essential that the Capsule server cut off this channel. First, Storage Capsules must have a fixed size. This imposes a small usability limitation, which we believe is reasonable given that other popular systems like TrueCrypt [31] fix the size of encrypted file containers. Enforcing the next constraint required to cut off storage channels is slightly more complicated. No matter what changes the primary VM makes to the Storage Capsule in secure mode, it must not be able to deduce what has been changed after the Capsule server exports the Storage Capsule. As discussed earlier, XTS-AES encryption with a different tweak value for each export satisfies this requirement. Whether the primary VM changes every single byte or does not touch anything, the resulting exported Storage Capsule will be random with respect to its original contents.

Social Engineering Attacks

If the primary virtual machine cannot find a way to leak data directly, then it can resort to influencing user behavior. The most straightforward example of a social engineering attack would be for the primary VM to deny service to the user by crashing at a specific time, and then measuring transition time back to normal mode. There is a pretty good chance that the user would respond to a crash by switching back to normal mode immediately, especially if the system is prone to crash-

ing under normal circumstances. In this case, the user may not even realize that an attack is taking place. Another attack that is higher-bandwidth, but perhaps more suspicious, would be for the primary VM to display a message in secure mode that asks the user to perform a task that leaks information. For example, a message could read “Automatic update failed, please open the update dialog and enter last scan time ‘4:52 PM’ when internet connectivity is restored.” Users who do not understand covert channels could easily fall victim to this attack. In general, social engineering is difficult to prevent. The Capsule system currently does not include any counter-measures to social engineering. In a real deployment, the best method of fighting covert channels would be to properly educate the users.

6.2 External Device Channels

Any device that is connected to a computer could potentially store information. Fortunately, most devices in a virtual machine are virtual devices, including the keyboard, mouse, network card, display, and disk. In a traditional system, two processes that have access to the keyboard could leak data through the caps-, num-, and scroll-lock state. The VMware VMM resets this device state when reverting to a snapshot, so a virtual machine cannot use it for leaking data. We did not test virtualization software other than VMware to see how it resets virtual device state.

Some optional devices may be available to virtual machines. These include floppy drives, CD-ROM drives, sound adapters, parallel ports, serial ports, SCSI devices, and USB devices. In general, there is no way of stopping a VM that is allowed to access these devices from leaking data. Even devices that appear to be read-only, such as a CD-ROM drive, may be able to store information. A VM could eject the drive or position the laser lens in a particular spot right before switching back to normal mode. While these channels would be easy to mitigate by adding noise, the problem worsens when considering a generic bus like USB. A USB device could store anything or be anything, including a disk drive. One could allow access to truly read-only devices, but each device would have to be examined on an individual basis to uncover covert channels. The Capsule system prevents these covert channels because the primary VM is not given access to external devices. If the primary VM needs access to external devices, then they would have to be disabled during secure mode.

6.3 External Network Channels

In addition to channels from the Primary VM in secure mode to normal mode, it is also important to consider channels between the Storage Capsule system and external machines during secure mode. If malware can utilize so many resources that it affects how responsive the VMM is to external queries (such as pings), then it can leak data to a colluding external computer.

The best way to mitigate external network channels is for the VMM to immediately drop all incoming packets with a firewall, not even responding with reset packets for failed connections. If the VMM does not require any connections during secure mode, which it did not for our implementation, then this is the easiest and most effective approach.

6.4 Virtual Machine Monitor Channels

In a virtualization system, everything is governed by the virtual machine monitor, including memory mapping, device I/O, networking, and snapshot saving/restoration. The VMM’s behavior can potentially open up new covert channels that are not present in a standard operating system. These covert channels are implementation-dependent and may or may not be present in different VMMs. This section serves as a starting point for thinking about covert channels in virtual machine monitors.

Memory Paging

Virtual machines are presented with a virtual view of their physical memory. From a VM’s perspective, it has access to a contiguous “physical” memory segment with a fixed size. When a VM references its memory, the VMM takes care of mapping that reference to a real physical page, which is commonly called a machine page. There are a few different ways that a VMM can implement this mapping. First, it could directly pin all of the virtual machine’s physical pages to machine pages. If the VMM uses this strategy, and it keeps the page mapping constant during secure mode and after restoration, then there is no way for a virtual machine to affect physical memory layout. However, this fixed mapping strategy is not always the most efficient way to manage memory.

Prior research describes resource management strategies in which the VMM may over-commit memory to virtual machines and page some of the VM’s “physical” memory out to disk [11, 32]. If the VMM employs this strategy, then a virtual machine can affect the VMM’s page table by touching different pages within its address space. The residual effects of page table manipulation may be visible to a VM after a snapshot restoration, unless the VMM first pages in all of the VM’s memory.

A snapshot restoration should page in all of a VM's memory in most cases. But, if it is a "background" restoration, then accessing a memory location that has not been loaded from the snapshot yet *and* is paged out to disk might incur two page faults instead of one, which can be measured and may leak information.

VMware ESX server employs a number of tricks that allow it to over-commit memory to virtual machines [32]. Each of the mechanisms described by Waldspurger allow efficient server consolidation, but also create an opportunity for covert channels. The first mechanism he describes is *ballooning*. Guest virtual machines contain balloon drivers that allocate extra memory and then tell the VMM it can page out the corresponding "physical" pages. With ballooning, a misbehaving guest OS can touch the balloon driver's pages and cause the VMM to load them in from disk. When the VM is subsequently reverted to a snapshot, the compromised OS can again touch all of the balloon driver pages to see whether they have already been faulted in prior to the snapshot restoration. The VMM could mitigate this ballooning channel by restoring the state of its page table to match the original set of ballooned pages when restoring a snapshot. Of course, ballooning may be used as a covert channel between simultaneously executing VMs in systems other than Capsule.

There has also been research on sharing memory pages with the same content in virtual machines [3, 32]. If the VMM maps multiple pages to one machine page and marks them as copy-on-write, then the virtual machine will encounter a page fault when writing to one of them. If a VM sets a large number of pages to have the same value during secure mode, then a much larger number of page faults will take place when restoring a snapshot. However, these page faults will only be noticeable to the VM if execution resumes in normal mode before all of the VM's memory pages are loaded from the snapshot file. In this case, the VM can measure the total restoration time or write to pages before they have loaded and test for page faults to recover information. If the VM resumes execution after its memory has been fully restored and pages have been re-scanned for duplication, then this covert channel will not work.

The Capsule system does not over-commit memory for virtual machines, so the memory saving techniques mentioned above do not take effect. Our implementation of the Capsule system does not employ any counter-measures to covert channels based on memory paging.

Virtual Networks

The Capsule system blocks external network access during secure mode, but it relies on a virtual network for communication between the secure VM and the primary VM. While the virtual network itself is stateless (to the best of our knowledge), anything connected to the network could potentially be a target for relaying information from secure mode to normal mode. The DHCP and NAT services in the VMM are of particular interest. A compromised virtual machine may send arbitrary packets to these services in an attempt to affect their state. For example, a VM might be able to claim several IP addresses with different spoofed MAC addresses. It could then send ARP requests to the DHCP service following snapshot restoration to retrieve the spoofed MAC addresses, which contain arbitrary data. The Capsule system restarts both the DHCP and NAT services when switching back to normal mode to avert this covert channel.

Any system that allows both a high-security and low-security VM to talk to a third trusted VM (the secure VM in Capsule) exposes itself another covert channel. Naturally, all bets are off if the primary VM can compromise the secure VM. Even assuming the secure VM is not vulnerable, the primary VM may still be able to convince it to relay data from secure mode to normal mode. Like the DHCP service on the host, the secure VM's network stack stores information. For example, the primary VM could send out TCP SYN packets with specific source port numbers that contain several bits of data right before reverting to normal mode. When the primary VM resumes execution, it could see the source ports in SYN/ACK packets from the secure VM.

It is unclear exactly how much data can be stashed in the network stack on an unsuspecting machine and how long that information will persist. The only way to guarantee that a machine will not inadvertently relay state over the network is to reboot it. This is the approach we take to flush the secure VM's network stack state when switching back to normal mode in Capsule.

Guest Additions

The VMware VMM supports additional software that can run inside of virtual machines to enhance the virtualization experience. The features of guest additions include drag-and-drop between VMs and a shared clipboard. These additional features would undermine the security of any virtual machine system with multiple confidentiality levels and are disabled in the Capsule system.

6.5 Core Hardware Channels

Core hardware channels allow covert communication via one of the required primary devices: CPU or disk. Memory is a core device, but memory mapping is handled by the VMM, and is discussed in the previous section. Core hardware channels might exist in any multi-level secure system and are not specific to Storage Capsules or virtual machines. One difference between prior research and this work is that prior research focuses on a threat model of two processes that are executing simultaneously on the same hardware. In the Capsule system, the concern is not with simultaneous processes, but with a low-security process (normal-mode VM) executing on the same hardware after a high-security process (secure-mode VM) has terminated. This constraint rules out some traditional covert channels that rely on resource contention, such as a CPU utilization channel.

CPU State

Restoring a virtual machine's state from a snapshot will overwrite all of the CPU register values. However, modern processors are complex and store information in a variety of persistent locations other than architecture registers. Many of these storage areas, such as branch prediction tables, are not well-documented or exposed directly to the operating system. The primary method for extracting this state is to execute instructions that take a variable number of clock cycles depending on the state and measure their execution time, or exploit speculative execution feedback. Prior research describes how one can use these methods to leak information through cache misses [24, 33].

There are a number of counter-measures to covert communication through CPU state on modern processors. In general, the more instructions that execute in between secure mode and normal mode, the less state will persist. Because the internal state of a microprocessor is not completely documented, it is unclear exactly how much code would need to run to eliminate all CPU state. One guaranteed method of wiping out all CPU state is to power off the processor. However, recent research on cold boot attacks [12] shows that it may take several minutes for memory to fully discharge. This strategy would lead to an unreasonably long delay when switching from secure mode to normal mode.

The ideal solution for eliminating covert CPU state channels in Capsule and other virtualization systems would be with hardware support. The latest CPUs already support hardware virtualization, which allows them to fully emulate instruction sets for virtual machines. An additional mechanism is needed when

switching between virtual machines that not only restores register and memory mappings, but also restores all state that could affect VM execution. This operation would load all of the cache data (L1, L2, and instruction), the branch prediction table, and any other intermediate state. It would also be sufficient to provide an instruction that would erase all of this data.

Although the Capsule system does not take counter-measures to prevent CPU state covert channels, the VMM restores a significant portion of the primary VM's original memory state before the VM can resume execution. This restoration will likely eliminate any residual cache lines from secure mode. It will also add a great deal of noise to any other CPU state, such as branch predictor tables, due to the number of instructions that execute during restoration. Computing the maximum bandwidth of CPU state channels in the Capsule system is future work.

Disk State

The caching and layout of files on disk, and in the file system, can be used to transmit information. Another covert channel exists in the positioning of the disk arm [10]. If the primary VM can manipulate disk state in a way that is readable after transitioning back to normal mode, then it can leak data. There are a few aspects of the Capsule system's design that make such an attack much more difficult. Any writes to the virtual disk during secure mode actually go to flat difference file that stores changes since the last snapshot operation. The primary VM has no control over where the data goes on disk, only how much is written, which would make exploitation of such a channel difficult. The VMM deletes this file when transitioning back to normal mode. This frees up any blocks that were allocated for the difference file. Finally, the VMM reads in the primary VM's original memory state from disk during snapshot restoration. This moves the position of the disk arm and adds considerable noise to the cache.

Despite the noise from transitioning back to normal mode, the disk cache is the most likely candidate for a covert channel. All writes are sent to a difference file and deleted, but reads touch the same virtual disk file during secure mode and normal mode. Consider the primary VM sequentially reading every block on its disk during secure mode to transmit a 1, or reading very little to transmit a 0. The primary VM might be able to tell what happened in secure mode by reading back disk blocks and measuring if they hit the disk cache. One would need to flush all of the disk caches to eliminate this channel. The Capsule system does not take any steps to mitigate disk state channels.

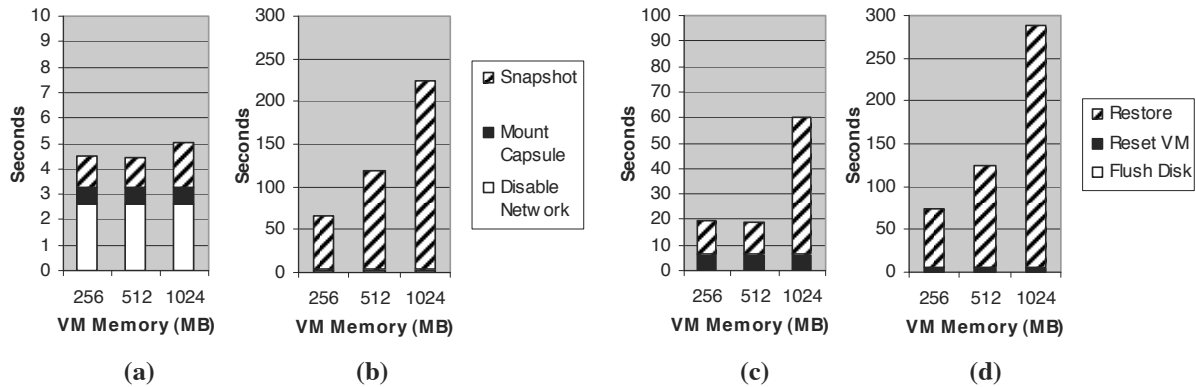


Figure 2. Transition times for different amounts of primary VM memory.

(a) to secure mode with background snapshot, (b) to secure mode with full snapshot

(c) to normal mode with background restore, and (d) to normal mode with full restore.

6.6 Mitigating VMM and Core Hardware Covert Channels

The design of Storage Capsules centers around improving local file encryption with a minimal impact on existing behavior. The user only has to take a few additional steps, and no new hardware is required. The current implementation is designed to guard against many covert channels, but does not stop leakage through all of them, such as the CPU state, through which data may leak from secure to normal mode. If the cost of small leaks outweighs usability and the cost of extra hardware, then there is an alternative design that can provide additional security.

One way of cutting off almost all covert channels would be to migrate the primary VM to a new isolated computer upon entering secure mode. This way, the virtual machine would be running on different core hardware and a different VMM while in secure mode, thus cutting off covert channels at those layers. VMware ESX server already supports live migration, whereby a virtual machine can switch from one physical computer to another without stopping execution. The user would have two computers at his or her desk, and use one for running the primary VM in secure mode, and the other for normal mode. When the user is done accessing a Storage Capsule, the secure mode computer would reboot and then make the Storage Capsule available for export over the network. This extension of the Capsule system's design would drastically reduce the overall threat of covert channels, but would require additional hardware and could add usability impediments that would not be suitable in many environments.

7. Performance Evaluation

There are three aspects of performance that are important for Storage Capsules: (1) transition time to secure mode, (2) system performance in secure mode, and (3) transition time to normal mode. It is important for transitions to impose only minimal wait time on the user and for performance during secure mode to be comparable to that of a standard computer for common tasks. This section evaluates Storage Capsule performance for transitions and during secure mode. The experiments were conducted on a personal laptop with a 2 Ghz Intel T2500 processor, 2 GB of RAM, and a 5200 RPM hard drive. Both the host and guest operating systems (for the secure VM and primary VM) were Windows XP Service Pack 3, and the VMM software was VMware Workstation ACE Edition 6.0.4. The secure VM and the primary VM were both configured with 512 MB of RAM and to utilize two processors, except where indicated otherwise.

The actual size of the Storage Capsule does not affect any of the performance numbers in this section. It does, however, influence how long it takes to run an import or export. Both import and export operations are expected to be relatively rare in most cases – import only occurs when loading a Storage Capsule from an external location, and export is required only when sending a Storage Capsule to another user or machine. Importing and exporting consist of a disk read, encryption (for export only), a local network transfer, and a disk write. On our test system, the primary VM could import a 256 MB Storage Capsule in approximately 45 seconds and export it in approximately 65 seconds. Storage Capsules that are imported and exported more often, such as e-mail attachments, are likely to be much smaller and should take only a few seconds.

7.1 Transitioning to and from Secure Mode

The transitions to and from secure mode consist of several tasks. These include disabling/enabling device output, mounting/dismounting the Storage Capsule, saving/restoring snapshots, waiting for an escape sequence, and obtaining the encryption key. Fortunately, some operations can happen in parallel. During the transition to secure mode, the system can do other things while waiting for user input. The evaluation does not count this time, but it will reduce the delay experienced by the user in a real deployment. VMware also supports both background snapshots (copy-on-write) and background restores (copy-on-read). This means that execution may resume in the primary VM before memory has been fully saved or restored from the snapshot file. The system will run slightly slower at first due to page faults, but will speed up as the snapshot or restore operation nears completion. A background snapshot or restore must complete before another snapshot or restore operation can begin. This means that even if the primary VM is immediately usable in secure mode, the system cannot revert to normal mode until the snapshot is finished.

Figure 2 shows the amount of time required for transitioning to and from secure mode with different amounts of RAM in the primary VM. Background snapshots and restorations make a huge difference. Transitioning to secure mode takes 4 to 5 seconds with a background snapshot, and 60 to 230 seconds without. The time required for background snapshots, mounting the Storage Capsule, and disabling network output also stays fairly constant with respect to primary VM memory size. However, the full snapshot time scales linearly with the amount of memory. Note that the user must wait for the full snapshot time before reverting to normal mode.

The experiments show that reverting to normal mode is a more costly operation than switching to secure mode, especially when comparing the background restore to the background snapshot operation. This is because VMware allows a virtual machine to resume immediately during a background snapshot, but waits until a certain percentage of memory has been loaded in a background restore. Presumably, memory reads are more common than memory writes, so copy-on-read for the restore has worse performance than copy-on-write for the snapshot. VMware also appears to employ a non-linear strategy for deciding what portion of a background restore must complete before the VM may resume execution. It waited approximately the same amount of time when a VM had 256 MB or 512 MB of RAM, but delayed significantly longer for the 1 GB case.

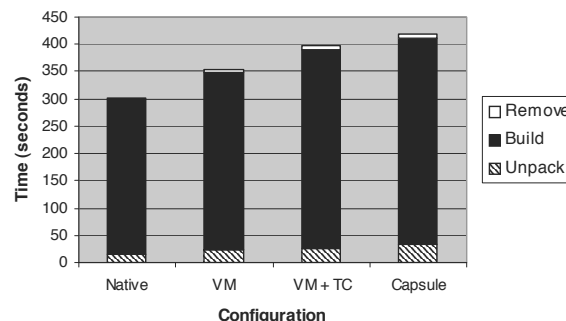


Figure 3. Results from building Apache with a native OS, a virtual machine, a virtual machine running TrueCrypt, and Capsule. Storage Capsules add only a 5% overhead compared to a VM with TrueCrypt, 18% slower than a plain VM, and 38% overhead compared to a native OS.

The total transition times to secure mode are all reasonable. Many applications will take 4 or 5 seconds to load a document anyway, so this wait time imposes little burden on the user. The transition times back to normal mode for 256 MB and 512 MB are also reasonable. Waiting less than 20 seconds does not significantly disrupt the flow of work. However, 60 seconds may be long wait time for some users. It may be possible to optimize snapshot restoration by using copy-on-write memory while the primary VM is in secure mode. This way, the original memory would stay in tact and the VMM would only need to discard changes when transitioning to normal mode. Optimizing transition times in this manner is future work.

7.2 Performance in Secure Mode

Accessing a Storage Capsule imposes some overhead compared to a normal disk. A Storage Capsule read or write request traverses the file system in the primary VM, and is then sent to the secure VM over the virtual network. The request then travels through a layer of encryption on the secure VM, out to its virtual disk, and then to the physical drive. We compared the disk and processing performance of Storage Capsules to three other configurations. These configurations consisted of a native operating system, a virtual machine, and a virtual machine with a TrueCrypt encrypted file container. For the evaluation, we ran an Apache build benchmark. This benchmark involves decompressing and extracting the Apache web server source code, building the code, and then removing all of the files. The Apache build benchmark probably represents the worst case scenario for Storage Capsule usage. We expect that the primary use of Storage Capsules will be for less disk-intensive activities like editing documents or images, for which the overhead should be unnoticeable.

Figure 3 shows the results of the Apache build benchmark. Storage Capsules performed well overall, only running 38% slower than a native system. Compared to a single virtual machine running similar encryption software (TrueCrypt), Storage Capsules add an overhead of only 5.1% in the overall benchmark and 31% in the unpack phase. This shows that transferring reads and writes over the virtual network to another VM has a reasonably small performance penalty. The most significant difference can be seen in the remove phase of the benchmark. It executes in 1.9 seconds on a native system, while taking 5.5 seconds on a VM, 6.5 seconds on a VM with TrueCrypt, and 7.1 seconds with Storage Capsules. The results from the VM and VM with TrueCrypt tests show, however, that the slowdown during the remove phase is due primarily to disk performance limitations in virtual machines rather than the Capsule system itself.

8. Conclusion and Future Work

This paper introduced Storage Capsules, a new mechanism for securing files on a personal computer. Storage Capsules are similar to existing encrypted file containers, but protect sensitive data from malicious software during decryption and editing. The Capsule system provides this protection by isolating the user's primary operating system in a virtual machine. The Capsule system turns off the primary OS's device output while it is accessing confidential files, and reverts its state to a snapshot taken prior to editing when it is finished. One major benefit of Storage Capsules is that they work with current applications running on commodity operating systems.

Covert channels are a serious concern for Storage Capsules. This research explores covert channels at the hardware layer, at the VMM layer, in external devices, and in the Capsule system itself. It looks at both new and previously examined covert channels from a novel perspective, because Storage Capsules have different properties than side-by-side processes in a traditional multi-level secure system. The research also suggests ways of mitigating covert channels and highlights their usability and performance trade-offs. Finally, we evaluated the overhead of Storage Capsules compared to both a native system and standard virtual machines. We found that transitions to and from secure mode were reasonably fast, taking 5 seconds and 20 seconds, respectively. Storage Capsules also performed well in an Apache build benchmark, adding 38% overhead compared to a native OS, but only a 5% penalty when compared to running current encryption software inside of a virtual machine.

In the future, we plan to further explore covert channels discussed in this work. This includes measuring their severity and quantifying the effectiveness of mitigation strategies. We also hope to conduct a study on usability of keyboard escape sequences for security applications. Storage Capsules rely on escape sequences to prevent spoofing attacks by malicious software, and it would be beneficial to know how many users of the Capsule system would still be vulnerable to such attacks.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. 0705672. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation

References

- [1] M. Blaze. A Cryptographic File System for UNIX. In *Proc. of the 1st ACM Conference on Computer and Communications Security*, Nov. 1993.
- [2] R. Browne. An Entropy Conservation Law for Testing the Completeness of Covert Channel Analysis. In *Proc. of the 2nd ACM Conference on Computer and Communication Security (CCS)*, Nov. 1994.
- [3] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. *ACM Transactions on Computer Systems*, 15(4), Nov. 1997.
- [4] A. Czeskis, D. St. Hilair, K. Koscher, S. Gribble, and T. Kohno. Defeating Encrypted and Deniable File Systems: TrueCrypt v5.1a and the Case of the Tattling OS and Applications. In *Proc. of the 3rd USENIX Workshop on Hot Topics in Security (HOTSEC '08)*, Aug. 2008.
- [5] M. Delio. Linux: Fewer Bugs than Rivals. *Wired Magazine*, <http://www.wired.com/software/coolapps/news/2004/12/66022>, Dec. 2004.
- [6] D. Denning and P. Denning. Certification of Programs for Secure Information Flow. *Communications of the ACM*, 20(7), Jul. 1977.
- [7] C. Fruhwirth. LUKS – Linux Unified Key Setup. <http://code.google.com/p/cryptsetup/>, Jan. 2009.
- [8] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, D. Boneh. Terra: a Virtual Machine-based Platform for Trusted Computing. In *Proc. of the 19th*

- ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2003.
- [9] T. Garfinkel and M. Rosenblum. When Virtual is Harder than Real: Security Challenges in Virtual Machine Based Computing Environments. In *Proc. of the 10th Workshop on Hot Topics in Operating Systems*, Jun. 2005.
 - [10] B. Gold, R. Linde, R. Peeler, M. Schaefer, J. Scheid, and P. Ward. A Security Retrofit of VM/370. In *AFIPS Proc., 1979 National Computer Conference*, 1979.
 - [11] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum. Cellular Disco: Resource Management Using Virtual Clusters on Shared-Memory Multiprocessors. In *Proc. of the Symposium on Operating System Principles*, Dec. 1999.
 - [12] J. Halderman, S. Schoen, N. Heninger, W. Clarkson, W. Paul, J. Calandrino, A. Feldman, J. Appelbaum, and E. Felten. Lest We Remember: Cold Boot Attacks on Encryption Keys. In *Proc. of 17th USENIX Security Symposium*, Jul. 2008.
 - [13] IEEE Computer Society. IEEE Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices. *IEEE Std 1619-2007*, Apr. 2008.
 - [14] O. Lagerkvist. ImDisk Virtual Disk Driver. <http://www.ltr-data.se/opencode.html#ImDisk>, Dec. 2008.
 - [15] T. Jaeger, R. Sailer, and X. Zhang. Analyzing Integrity Protection in the SELinux Example Policy. In *Proc. of the 12th USENIX Security Symposium*, Aug. 2003.
 - [16] M. Kang and I. Moskowitz. A Pump for Rapid, Reliable, Secure Communication. In *Proc. of the 1st ACM Conference on Computer and Communication Security (CCS)*, Nov. 1993.
 - [17] R. Kemmerer. An Approach to Identifying Storage and Timing Channels. In *ACM Transactions on Computer Systems*, 1(3), Aug. 1983.
 - [18] M. Liskov, R. Rivest, and D. Wagner. Tweakable Block Ciphers. In *Advances in Cryptology – CRYPTO ’02*, 2002.
 - [19] R. Meushaw and D. Simard. NetTop: Commercial Technology in High Assurance Applications. <http://www.vmware.com/pdf/Tech-TrendNotes.pdf>, 2000.
 - [20] Microsoft Corporation. BitLocker Drive Encryption: Technical Overview. <http://technet.microsoft.com/en-us/library/cc732774.aspx>, Jan. 2009.
 - [21] I. Moskowitz and A. Miller. Simple Timing Channels. In *Proc. of the IEEE Symposium on Security and Privacy*, May 1994.
 - [22] A. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *Proc. of the 26th ACM Symposium on Principles of Programming Languages (POPL)*, Jan. 1999.
 - [23] National Security Agency. Security-enhanced Linux. <http://www.nsa.gov/selinux>, Jan. 2008.
 - [24] C. Percival. Cache Missing for Fun and Profit. In *Proc. of BSDCan 2005*, May 2005.
 - [25] A. Roshal. WinRAR Archiver, a Powerful Tool to Process RAR and ZIP Files. <http://www.rarlab.com/>, Jan. 2009.
 - [26] R. Sailer, X. Zhang, T. Jaeger, and L. Van Doorn. Design and Implementation of a TCG-Based Integrity Measurement Architecture. In *Proc. of the 13th USENIX Security Symposium*, Aug. 2004.
 - [27] Secunia. Xen 3.x – Vulnerability Report. <http://secunia.com/product/15863/?task=statistics>, Jan. 2009.
 - [28] Secunia. Linux Kernel 2.6.x – Vulnerability Report. <http://secunia.com/product/2719/?task=statistics>, Jan. 2009.
 - [29] Trusted Computing Group. Trusted Platform Module Main Specification. <http://www.trustedcomputinggroup.org>, Ver. 1.2, Rev. 94, June 2006.
 - [30] J. Trostle. Multiple Trojan Horse Systems and Covert Channel Analysis. In *Proc. of Computer Security Foundations Workshop IV*, Jun. 1991.
 - [31] TrueCrypt Foundation. TrueCrypt – Free Open-Source On-the-fly Encryption. www.truecrypt.org, Jan. 2009.
 - [32] C. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proc. of the 5th Symposium on Operating Systems Design and Implementation*, Dec. 2002.
 - [33] Z. Wang and R. Lee. Covert and Side Channels Due to Processor Architecture. In *Proc. of the 22nd Annual Computer Security Applications Conference (ACSAC)*, Dec. 2006.
 - [34] T. Weber. Criminals ‘May Overwhelm the Web’. *BBC News*, <http://news.bbc.co.uk/1/hi/business/6298641.stm>, Jan. 2007.
 - [35] WinZip International LLC. WinZip – The Zip File Utility for Windows. <http://www.winzip.com/>, Jan. 2009.
 - [36] XenSource, Inc. Xen Community. <http://xen.xensource.com/>, Apr. 2008.

Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms

Ralf Hund Thorsten Holz Felix C. Freiling

Laboratory for Dependable Distributed Systems

University of Mannheim, Germany

hund@uni-mannheim.de, {holz, freiling}@informatik.uni-mannheim.de

Abstract

Protecting the kernel of an operating system against attacks, especially injection of malicious code, is an important factor for implementing secure operating systems. Several kernel integrity protection mechanisms were proposed recently that all have a particular shortcoming: They cannot protect against attacks in which the attacker re-uses existing code within the kernel to perform malicious computations. In this paper, we present the design and implementation of a system that fully automates the process of constructing instruction sequences that can be used by an attacker for malicious computations. We evaluate the system on different commodity operating systems and show the portability and universality of our approach. Finally, we describe the implementation of a practical attack that can bypass existing kernel integrity protection mechanisms.

1 Introduction

Motivation. Since it is hard to prevent users from running arbitrary programs within their own account, all modern operating systems implement protection concepts that protect the realm of one user from another. Furthermore, it is necessary to protect the kernel itself from attacks. The basis for such mechanisms is usually called *reference monitor* [2]. A reference monitor controls all accesses to system resources and only grants them if they are allowed. While reference monitors are an integral part of any of today's mainstream operating systems, they are of limited use: because of the sheer size of a mainstream kernel, the probability that some system call, kernel driver or kernel module contains a vulnerability rises. Such vulnerabilities can be exploited to subvert the operating system in arbitrary ways, giving rise to so called *rootkits*, malicious software running without the user's notice.

In recent years, several mechanisms to protect the integrity of the kernel were introduced [6, 9, 15, 19, 22], as we now explain. The main idea behind all of these approaches is that the memory of the kernel should be protected against unauthorized injection of code, such as rootkits. Note that we focus in this work on kernel integrity protection mechanisms and not on control-flow integrity [1, 7, 14, 18] or data-flow integrity [5] mechanisms, which are orthogonal to the techniques we describe in the following.

1.1 Kernel Integrity Protection Mechanisms

Kernel Module Signing. Kernel module signing is a simple approach to achieve kernel code integrity. When kernel module signing is enabled, every kernel module should contain an embedded, valid digital signature that can be checked against a trusted root certification authority (CA). If this check fails, loading of the code fails, too. This technique has been implemented most notably for Windows operating systems since XP [15] and is used in every new Windows system.

Kernel module signing allows to establish basic security guidelines that have to be followed by kernel code software developers. But the security of the approach rests on the assumption that the already loaded kernel code, i.e., the kernel and *all* of its modules, does not have a vulnerability which allows for execution of unsigned kernel code. It is thus insufficient to check for kernel code integrity only upon loading.

W \oplus X. W \oplus X is a general approach which aims at preventing the exploitation of software vulnerabilities at runtime. The idea is to prevent execution of injected code by enforcing the W \oplus X property on all, or certain, page tables of the virtual address space: A memory page must never be writable *and* executable at the same time. Since injected code execution always implies previous

instruction writes in memory, the integrity of the code can be guaranteed. The $W \oplus X$ technique first appeared in OpenBSD 3.3; similar implementations are available for other operating systems, including the PaX [28] and Exec Shield patches for Linux, and PaX for NetBSD. Data Execution Prevention (DEP) [16] is a technology from Microsoft that relies on $W \oplus X$ for preventing exploitation of software vulnerabilities and has been implemented since Windows XP Service Pack 2 and Windows Server 2003.

The effectiveness of $W \oplus X$ relies on the assumption that the attacker wishes to modify and execute code in kernel space. In practice, however, an attacker usually first gains userspace access which implies the possibility to alter page-wise permission in the userspace portion of the virtual address space. Due to the fact that the no-executable bit in the page-table is not fine-grained enough, it is not possible to mark a memory page to be executable *only* in user mode. So an attacker may simply prepare her instructions in userspace and let the vulnerable code jump there.

NICKLE. *NICKLE* [19] is a system which allows for lifetime kernel code integrity, and thus rootkit prevention, by exploiting a technology called *memory shadowing*. *NICKLE* is implemented as virtual machine monitor (VMM) which maintains a separate so-called *shadow memory*. The shadow memory is not accessible from within the VM guest and contains copies of certain portions of the VM guest's main memory. Newly executing code, i.e., code that is executed for the first time, is authenticated using a simple cryptographic hash value comparison and then copied to the shadow memory transparently by *NICKLE*. Since the VMM is trusted in this model, it is guaranteed that no unauthenticated modifications to the shadow memory can be applied as executing guest code can never access the privileged shadow memory. Therefore, any attempt to execute unauthenticated code can be foiled in the first place. Another positive aspect of this approach is that it can be implemented in a rather generic fashion, meaning that it is perfectly applicable to both open source and commodity operating systems. Of course, *NICKLE* itself has to make certain assumptions about underlying file format of executable code, e.g., driver files, since it needs to understand the loading of these files. Currently, *NICKLE* supports Windows 2000, Windows XP, as well as Linux 2.4 and 2.6 based kernels. So far, *NICKLE* has been implemented for QEMU, VMware, and VirtualBox hypervisors. The QEMU source code is publicly available [20].

The isolation of the VMM from the VM Guest allows for a comparably unrestrictive threat model. In the given system, an attacker may have gained the highest level of privilege within the VM guest and may access the entire memory space of the VM. In other words, an ad-

versary may compromise arbitrary system entities, e.g., files, processes, etc., as long as the compromise happens only *inside* the VM.

SecVisor. SecVisor [22] is a software solution that consist of a general, operating system independent approach to enforce $W \oplus X$ based on a hypervisor and memory virtualization. In the threat model for SecVisor an attacker can control everything but the CPU, the memory controller, and kernel memory. Furthermore, an attacker can have the knowledge of kernel exploits, i.e., she can exploit a vulnerability in kernel mode. In this setting, SecVisor “protects the kernel against code injection attacks such as kernel rootkits” [22]. This is achieved by implementing a hypervisor that restricts what code can be executed by a (modified) Linux kernel. The hypervisor virtualizes the physical memory and the MMU to set page-table-based memory protections. Furthermore, SecVisor verifies certain properties on kernel mode entry and exit, e.g., all kernel mode exits set the privilege level of the CPU to that of user mode or the instruction pointer points to approved code at kernel entry. Franklin et al. showed that these properties are prone to attacks and successfully injected code in a SecVisor-protected Linux kernel [8], but afterwards also corrected the errors found.

1.2 Bypassing Integrity Protection Mechanisms

Based on earlier programming techniques like *return-to-libc* [17, 21, 27], Shacham [23] introduced the technique of *return-oriented programming*. This technique allows to execute arbitrary programs in privileged mode without adding code to the kernel. Roughly speaking, it misuses the system stack to “re-use” existing code fragments (called *gadgets*) of the kernel (we explain this technique in more detail in Section 2). Shacham analyzed the GNU Linux libc of Fedora Core 4 on an Intel x86 machine and showed that executing one malicious instruction in system mode is sufficient to construct arbitrary computations from existing code. No malicious code is needed, so most of the integrity protection mechanisms fail to stop this kind of attack.

Buchanan et al. [4] recently extended the approach to the Sparc architecture. They investigated the Solaris 10 C library, extracted code gadgets and wrote a compiler that can produce Sparc machine programs that are made up entirely of the code from the identified gadgets. They concluded that it is not sufficient to prevent introduction of malicious *code*; we must rather prevent introduction of malicious *computations*.

Attacker Model. Like the mentioned literature, we base our work on the following reasonable attacker model. We assume that the attacker has full access to the user's address space in normal mode (local attacker) and that there exists at least one vulnerability within a system call such that it is possible to point the control flow to an address of the attacker's choice at least once while being in privileged mode. In practice, a vulnerable driver or kernel module is sufficient to satisfy these assumptions. Our attack model also covers the typical "remote compromise" attack scenario in network security where attackers first achieve local user access by guessing a weak password and then escalate privileges.

Contributions. In this paper, we take the obvious next step to show the futility of current kernel integrity protection techniques. We make the following research contributions:

- While previous work [4, 21, 23] was based on manual analysis of machine language code to create gadgets, we present a system that fully automates the process of constructing gadgets from kernel code and translating arbitrary programs into return-oriented programs. Our automatic system can use any kernel code (not only *libc*, but also drivers for example) even on commodity operating systems.
- Using our automatic system, we construct a portable rootkit for Windows systems that is entirely based on return-oriented-programming. It therefore is able to bypass even the most sophisticated integrity checking mechanism known today (for example NICKLE [19] or SecVisor [22]).
- We evaluate the performance of return-oriented programs and show that the runtime overhead of this programming technique is significant: In our tests we measured a slowdown factor of more than 100 times in sorting algorithms. However, for exploiting a system this slowdown might not be important.

Outline. The paper is structured as follows. In Section 2 we provide a brief introduction to the technique of return-oriented-programming. In Section 3 we introduce in detail our framework for automating the gadget construction and translating arbitrary programs into return-oriented programs. We present evaluation results for our framework in Section 4: Using ten different machines, we confirm the portability and universality of our approach. We present the design and implementation of a return-oriented rootkit in Section 5 and finally conclude the paper in Section 6 with a discussion of future work.

2 Background: Return-Oriented Programming

The idea behind a return-to-*libc* attack [17, 27] is that the attacker can use a buffer overflow to overwrite the return address on the stack with the address of a legitimate instruction which is located in a library, e.g., within the C runtime *libc* on UNIX-style systems. Furthermore, the attacker places the arguments to this function to another portion of the stack, similar to classical buffer overflow attacks. This approach can circumvent some buffer overflow protection techniques, e.g., non-executable stack.

The technique of return-oriented programming was introduced by Shacham et al. [4, 23]. It generalizes return-to-*libc* attacks by chaining short new instructions streams ("useful instructions") that then return. Several instructions can be combined to a *gadget*, the basic block within return-oriented programs that for example computes the AND of two operands or performs a comparison. Gadgets are self-contained and perform one well-defined step of a computation. The attacker uses these gadgets to cleverly craft stack frames that can then perform arbitrary computations. Fig. 1 illustrates the process of return-oriented programming. First, the attacker identifies useful instructions that are followed by a `ret` instruction (e.g., instruction sequences A, B and C in Fig. 1). These are then chained to gadgets to perform a certain operation. For example, instruction sequences A and B are chained together to gadget 1 in Fig. 1. On the stack, the attacker places the appropriate return addresses to these instruction sequences. In the example of Fig. `refig:rop` the return addresses on the stack will cause the executions of gadget 1 and then gadget 2. The stack pointer ESP determines which instruction to fetch and execute, i.e., within return-oriented programming the stack pointer adopts the role of the instruction pointer (IP): Note that the processor does not automatically increment the stack pointer, but the `ret` instruction at the end of each useful instruction does.

The authors showed that both the *libc* library of Linux running on the x86 architecture (CISC) as well as the *libc* library of Solaris running on a SPARC (RISC) contain enough useful instructions to construct meaningful gadgets. They manually analyzed the *libc* of both environments and constructed a library of gadgets that is Turing-complete. We extend their work by presenting the design and implementation of a fully automated return-oriented framework that can be used on commodity operating systems. Furthermore, we describe an actual attack against kernel integrity protection systems by implementing a return-oriented rootkit.

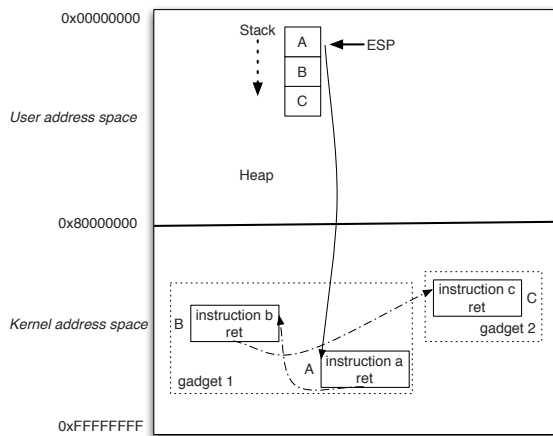


Figure 1: Schematic overview of return-oriented programming on the Windows platform

3 Automating Return-Oriented Programming

In order to be able to create and execute return-oriented programs in a generic way, we created our own, modular toolset which enables one to abstract from the varying concrete conditions one faces in this context. Additionally, our system greatly simplifies the development of return-oriented programs by providing high-level constructs to accomplish certain tasks. Figure 2 provides a schematic overview of our system; it is partitioned into three core components:

- **Constructor.** The Constructor scans a given set of files containing executable code, spots useful instruction sequences and builds return-oriented gadgets in an automatic fashion. These well-defined gadgets serve as a low-level abstraction and interface to the Compiler.
- **Compiler.** The Compiler provides a comparatively high-level language for programming in a return-oriented way. It takes the output of the Constructor along with a source file written in a dedicated language to produce the final memory image of the program.
- **Loader.** As the Compiler's output is position independent, it is the task of the Loader to resolve relative memory addresses to absolute addresses. This component is implemented as library that is supposed to be linked against by an exploit.

All components have been implemented in C++ and currently we support Windows NT-based operating systems running on an IA-32 architecture. In the following paragraphs, we give more details on each component's inner workings.

3.1 Automated Gadget Construction

One of the most essential parts of our system is the *automated construction* of return-oriented gadgets, thus enabling us to abstract from a concrete set of executable code being exploited for our purposes. This is in contrast to previous work [4, 23], which focused on concrete versions of a C library instead. Our system works on an arbitrary set of files containing valid x86 machine code instructions; we will henceforth refer to these files as the *codebase*.

Our framework implements the creation of gadgets in the so-called *Constructor*, which performs three subsequent jobs: First, it scans the codebase to find *useful instruction sequences*, i.e., instructions preceding a return (`ret`) instruction. These instructions can then be used to implement a return-oriented program by concatenating the sequences in a specific way. Our current implementation targets machines running an arbitrary Windows version as operating system and thus we use all driver executables and the kernel as codebase in the scanning phase. In the second step, our algorithm chains the instruction sequences together to form *gadgets* that perform basic operations. We define the term gadget analog to Shacham [23], i.e., gadgets comprise composite useful instruction sequences to accomplish a well-defined task (e.g., perform an AND operation or check a boolean condition). More precisely, when we talk of *concrete gadgets*, we mean the corresponding *stack allocation*, i.e., the contents (return-addresses, constants, etc.) of the memory area the stack register points to. Gadgets represent an intermediate abstraction layer whose elements are the basic units subsequently used by the Compiler for building the final return-oriented program. Gadgets being written to the Constructor's final output file are called *final gadgets*. In the third step, the Constructor searches for exported symbols in the codebase and saves these in the output file for later use by the Compiler.

3.1.1 Finding Useful Instruction Sequences

The first decision that has to be made is describing the basic instruction sequences being the very core of a return-oriented program. As previously mentioned, these instructions occur prior to a `ret` x86 assembler instruction. We have to decide how many instructions preceding a return are considered. For instance, the Constructor might look for sequences such as `mov eax, ecx; add eax, edx; ret`, and incorporate these in the subsequent gadget construction. An easier approach, however, is to consider only a single instruction before a return instruction. Of course, the former attempt has the advantage of being more comprehensive, along with the drawback of requiring additional overhead. This stems from the fact that one has to take every instruction's pos-

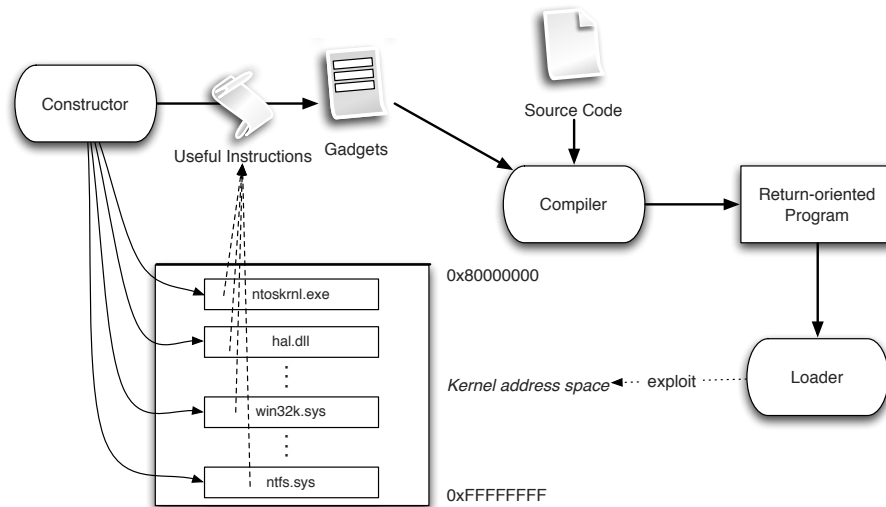


Figure 2: Schematic system overview

sible side-effects on registers and memory into account. In our work, we have thus chosen to implement the latter approach. Rudimentary research has shown that the additional value of using longer instruction sequences hardly justifies the imposed overhead since the effect of the former is not very significant in practice: We have observed that the high density of the x86 instructions encoding does not introduce substantial surplus concerning additional instruction sequences. We would also like to stress that this simplified approach has not turned out to be problematic in our work so far since the codebase of every system we evaluated held sufficient instruction sequences to implement arbitrary return-oriented programs (see Section 4 for details). However, our system might still be extended in the future in order to support more than one instruction preceding a return instruction.

To scan the codebase for useful instruction sequences, the Constructor first enumerates all sections of the PE file that contain executable code and scans these for x86 `ret` opcodes. In addition to the standard `ret` instruction, which has the opcode `0xC3`, we are also interested in return instructions that add an immediate value to the stack, represented by opcode `0xC2` and followed by the 16bit immediate offset. The former are favorable to the latter as they induce less memory consumption in the stack allocation since we need to append effectively unused memory before the next instruction.

Having found all available return instructions, the Constructor then byte-wise disassembles the sequence backwards, thereby building a trie. This works analogously to the method already described by Shacham [23]. In order to disassemble encoded x86 instructions, our program uses the *distorm* library [10].

3.1.2 Building Gadgets

The next logical step is chaining together instruction sequences to form structured gadgets that perform basic operations. Gadgets built by the Constructor form the very basic entities that are chained together by the Compiler for building the program stack allocation. Due to the clear separation of the Constructor and the Compiler, final gadgets are independent of each other. Therefore, each final gadget constitutes an autonomous piece of return-oriented code: Final gadgets take a set of source operands, perform a well-defined operation on these, and then write the result into a destination operand. In our model, source and destination operands are always memory variable addresses. For example, an addition gadget takes two source operands, i.e., memory addresses to both input variables, as input, adds both values, and then writes back the result to the memory address pointed to by the destination operand. There are certain exceptions to this rule, namely final gadgets that perform very specific tasks for certain situations, e.g., manipulating the stack register directly. Final gadgets are designed to be fine-grained with respect to the constraints imposed by the operand model. They can be separated into three classes: Arithmetic, logical and bitwise operations; control flow manipulations (static and dynamic); and stack register manipulations.

The crucial point in gadget construction concerns the algorithm that is deployed to spot appropriate useful instructions and the rules in which they are chained together. We consider completeness, memory consumption, and runtime to be the three dominating properties. By completeness, we mean the algorithm's fundamental ability to construct gadgets even in a minimal codebase,

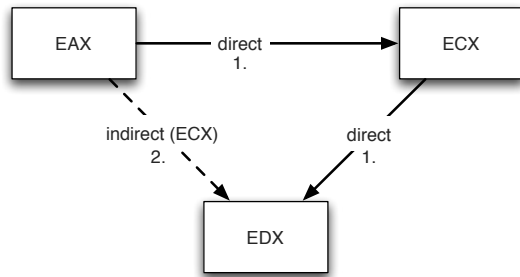


Figure 3: MOV connection graph: Chained instructions can be used to emulate other instructions.

where minimal indicates a codebase with a theoretically minimal set of instruction sequences to allow for corresponding gadget computations. By memory consumption, we denote that the constructed gadgets should be preferably small in size. By runtime, we mean that the algorithm should terminate within a reasonable period of time. Due to the CISC nature of x86 and the corresponding complexity of the machine language, we consider the completeness property to be the most difficult one to achieve. The many subtle details of this platform make it hard to find all possible combinations of useful instruction performing a given operation.

In the following, we provide a deeper look into our gadget construction algorithm. As with every modern CPU, x86 is a register-based architecture. This observation drives the starting point of our algorithm in that its first step is to define a set of general purpose registers that are allowed to be used, i.e., read from or written to, by gadget computations. This also has the positive side-effect that it enables an easy way to control which registers are modified by the return-oriented program. We will henceforth call these registers *working registers*.

Basic Gadgets. Starting from this point, we gradually construct lists of gadgets performing a related task for each working register. More precisely, the first step is to check which register can be loaded with fixed values, an operation that can easily be achieved with a `pop <register>; ret` sequence (*register-based constant load gadgets*). Afterwards, the Constructor searches for unary instructions sequences, e.g., `not` or `neg`, that take working registers as their operands (*register-based unary operation gadgets*). Subsequently, the algorithm checks which working registers are connected by binary instruction sequences, e.g., `mov`, `add`, `and`, and the like (*register-based binary operation gadgets*). In order to find indirectly connected registers, we build a directed graph for each operation whereas a node represents a register and an edge depicts an operation from the source register to the destination register

(also always being a source operand on x86). Then, we traverse all paths in the graph for each node. For example, let us assume the following situation: The given codebase allows for the execution of `mov ecx, eax; ret` and `mov edx, ecx; ret` sequences, but does not supply `mov edx, eax` sequences. We can easily find the corresponding path in our graph and hence construct a gadget that moves the content of `eax` to `edx` by chaining together both sequences (see Fig. 3). Since x86 is not a load-store-architecture, i.e., most instructions may take direct memory operands, we also search for memory operand based instructions (*register-based memory load-/operation gadgets*). This also allows us to check which working registers can be loaded with memory contents, for instance, `mov eax, [ecx]; ret` easily allows us to load an arbitrary memory location into `eax` by preparing `ecx` accordingly. The result of this first stage of the algorithm are lists of internal gadgets being bound to working registers and performing certain operations on these.

In the next stage, our algorithm merges working register-based gadgets to form new, final gadgets that perform certain operations, e.g., addition, multiplication, bitwise OR, and so on (*final unary/binary operation gadgets*). Therefore, it generates every possible combination of according register-based load/store and operation gadgets to choose the one being minimal with respect to consumed memory space. In the construction, we have to take into account possibly emerging side-effects when connecting instruction sequences. We say that a gadget has a *side-effect* on a given register when it is modified during execution. For instance, if we wish to build a gadget that loads two memory addresses into `eax` and `ecx` and appends an `and eax, ecx; ret` sequence, we have to make sure that both load gadgets do not have side-effects on each other's working register.

Control Flow Alteration Gadgets. Afterwards, the algorithm constructs final gadgets that allow for static and dynamic control flow alterations in a return-oriented program (*final comparison and dynamic control flow gadgets*). Therefore, we must first compare two operands with either a `cmp` or `sub` instruction, both have the same impact on the `eflags` registers which holds the condition flags. The main problem in this context is gaining access to the CPU's flag registers as this is only possible with a limited set of instructions. As already pointed out by Shacham [23], a straightforward solution is to search for the `lahf` instruction, which stores the lower part of the `eflags` register into `ah`. Another possibility is to search for `setCC` instructions, which store either one or zero depending on whether the condition is true or not. Thereby, `CC` can be any condition known to the CPU, e.g., equal, less than, greater or equal, and so on. Once

we have stored the result of the comparison (where 1 means true and 0 means false) the natural way to proceed is to multiply this value by four and add it to a jump table pointer. Then, we simply move the stack register to the value being pointed at.

Additional Gadgets. Finally, the Constructor builds some special gadgets that enable very specific tasks, such as, e.g., pointer dereferencing (*final dereferencing gadgets*), and direct stack or base register manipulation (*stack register manipulation gadgets*). The latter are required in certain situation as described in the next section. The final output of the Constructor is an XML file that describes the final gadgets along with a list of exported symbols from the codebase.

Turing Completeness. Gadgets are used within return-oriented programming as the basis blocks of each computation. An interesting question is now which kind of gadgets are needed such that return-oriented programming is *Turing complete*, i.e., it can compute every Turing-computable function [30]. We construct gadgets to load/store variables (including pointer dereferencing), branch instructions, and also gadgets for arithmetic operations (i.e., addition and not). This set of gadgets is minimal in the sense that we can construct from these gadgets any program: Our return-oriented framework can implement so called *GOTO languages*, which are Turing complete [12].

3.2 Compiler

The Compiler is the next building block of our return-oriented framework: This tool takes the final gadgets constructed by the Constructor along with a high-level language source file as input to produce the stack allocation for the return-oriented program. The Compiler acts as an abstraction of the concrete codebase so that developers do not have to mess with the intricacies of the codebase on the lowest layer; moreover, it provides a comparatively easy and abstract way to formulate a complex task to be realized in a return-oriented fashion. The Compiler's output describes the stack allocation as well as additional memory areas serving a specific purpose in a position independent way, i.e., it only contains relative memory addresses. This stems from the fact that the Compiler cannot be aware of the final code locations since drivers may be relocated in kernel memory due to address conflicts. Moreover, the program memory's base location may be unknown at this stage. It is hence the task of the Loader to resolve these relative addresses to absolute addresses (see next section).

3.2.1 Dedicated Language

Naturally, one of the first considerations in compiler development concerns the programming language employed. One possibility is to build the Compiler on top of an already existing language, ideally one that is designed to accomplish low-level tasks, such as C. However, this also introduces a profound overhead as all the language's peculiarities, e.g., the entire type system, must be implemented in a correct manner. Due to our very specific needs, we have found none of the existing language to be suited for our purpose and thus decided to create a dedicated language. It bears certain resemblance to many existing languages, specifically C. Our dedicated language provides the following code constructs:

- subroutines and recursive subroutine calls,
- a basic type-system that consists of two variable types,
- all arithmetic bitwise, logical and pointer operators known to the C language with some minor deviations, and
- nested dynamic control flow decisions and nested conditional looping.

Additionally, we also support external subroutine calls which enables one to dispatch operations to exported symbols from drivers or the kernel; this gives us more flexibility, greatly simplifies the development of return-oriented programs, and also substantially decreases stack allocation memory consumption.

Two basic variable types are supported: Integers and character arrays, the former being 32bit long while in case of the latter, strings are zero-terminated just as in C. Along with the ability to call external subroutines, this enables us to use standard C library functions exported by the kernel to process strings within the program. We do not need support for short and char integers for now as we do not consider these to be substantially relevant for our needs. Short integer operations thus must be emulated by the return-oriented program when needed.

The Compiler has been implemented in C++ using the ANTLR compiler generation framework [29]. Source code examples for our dedicated programming language are introduced in Section 5.4 and in Appendix B.

3.2.2 Memory Layout

Just as the Constructor chains together instruction sequences, the Compiler chains together gadgets to build a program performing the semantics imposed by the source code. Apart from that, it also defines the memory layout and assigns code and data to memory locations. By *code*, we henceforth mean the stack allocation of the

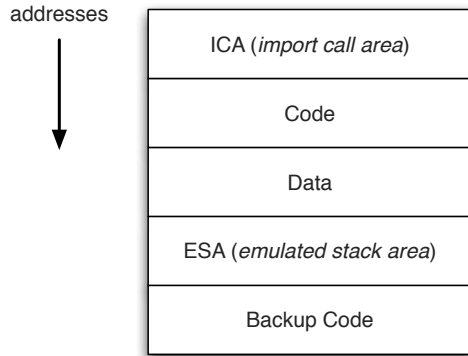


Figure 4: Memory layout of program image within our return-oriented framework

sum of all gadgets of a program (mostly return addresses to instruction sequences); this must not be confused with real CPU code, i.e., the code as we defined does not need *any* executable memory, but appears like usual data to the processor. This is the key concept in bypassing kernel integrity protection mechanisms: We do not need to inject code since we re-use existing code within the kernel during an exploit.

When we use the term *data*, we henceforth mean the memory area composed by the program’s variables and temporary internal data required by computations. We then constitute the memory layout to consist of a linear memory space we hereafter call the *program image*, which is shown in Fig. 4. Furthermore, some regions of this space serve special purposes we describe later on. In total, we separate the program image into five sections: Code, data, ICA, ESA and backup code.

The so-called *import call area* (ICA) resides at the very beginning, i.e., the lowest address, of the address space. When executing external function calls, the program prepares the call to be dispatched with the stack pointer `esp` pointing at the end (the highest address) of the ICA. Therefore, it first prepares this region by copying the arguments and return addresses to point to specific stack manipulation gadgets. Special care has to be taken concerning the imposed calling convention of the callee. We support both relevant conventions, namely `stdcall`, i.e., the callee cleans up the stack, and `cdecl`, i.e., the caller cleans up the stack. The need for such a dedicated section stems from the fact that, upon entry, the callee considers all memory regions below `esp` to be available to hold its local variables, hence overwriting return-oriented code that might still be needed at a later stage, i.e., when a jump back occurs.

Following the ICA, the Compiler places the code, i.e., return addresses and constant values to be popped into registers, followed by the data section which holds the

actual explicit variables as well as some implicit temporary variables that are mandatory during computation. After that, the *emulated stack area* (ESA) resides, which is used to emulate a “stack in the stack” to allow for recursive subroutine calls in the return-oriented program. The program image is terminated by an optional backup of the code section, a necessity that arises from a peculiarity of the Windows operating system we discuss later on in Section 5.2.

3.2.3 Miscellaneous

We also provide special language constructs enabling one to retransfer the CPU control flow to a non-return oriented source. For instance, in the typical case of an exploit and subsequent execution of return-oriented code, we might wish to return to the vulnerable code to allow for a continuation of execution. Therefore, we must restore the `esp` register to point to its original value. Our language hence provides appropriate primitives to tamper with the stack.

3.3 Loader

The final building block of our system consists of the Loader whose main task is to resolve the program image’s relative addresses to absolute addresses. Therefore, it must first enumerate all loaded drivers in the system and retrieve their base addresses. Luckily, Windows provides a function by the name of `EnumDeviceDrivers` that lets us accomplish this task even in usermode.

For the sake of flexibility, the Loader is implemented as a dynamic link library (DLL). The actual exploit transfers the task of building the final program image to the Loader and then adjusts the exploit to modify the instruction pointer `eip` to a gadget that modifies the stack (e.g., `pop esp; ret`) to start the execution of the return-oriented program. It is therefore sufficient for the exploit to be able to modify eight subsequent bytes in the stack frame: The first four bytes are a return address (of the sequence `pop esp; ret`) that is executed upon the next `ret` in the current control flow; the last four bytes point to the entry point of the program image to which control will flow after the execution of the next `ret`.

4 Evaluation Results

We implemented the system we described in the previous section in the C++ programming language. The Constructor consists of about 3,400 lines of code (LOC), whereas the Compiler is implemented in about 3,200 LOC. The loader only needs 700 LOC.

In the following, we present evaluation results for the individual components of our framework. We first show measurement results for the Constructor and Compiler and then provide several examples of the gadgets constructed by our tools. Finally we also measure the run-time overhead of return-oriented programs.

4.1 Constructor and Compiler

4.1.1 Evaluation of Useful Instructions and Gadget Construction

One goal of our work is to fully automate the process of constructing gadgets from kernel code on different platforms without the need of manual analysis of machine language code. We thus tested the Constructor on ten different machines running different versions of Windows as operating system: Windows 2003 Server, Windows XP, and Windows Vista were considered in different service pack versions to assess a wide variety of platforms. On each tested platform the Constructor was able to find enough useful instructions to construct all important gadgets that are needed by the Compiler, i.e., on each platform we are able to compile arbitrary return-oriented programs. This substantiates our claim that our framework is general and portable.

Table 1 provides an overview of the results for the gadget construction algorithm for six of the ten test configurations. We omitted the remaining four test results for the sake of brevity; the results for these machines are very similar to the listed ones. The table contains test results for two scenarios: On the one hand, we list the number of return instructions and trie leaves when using *any* kernel code, e.g., all drivers and kernel components. On the other hand, we list in the restricted column (res.) the results when using *only* the main kernel component (`ntoskrnl.exe`) and the Win32-subsystem (`win32k.sys`) for extracting useful instructions. These two components are available in any Windows environment and thus constitute a memory region an attacker can always use to build gadgets.

The number of return instructions found varies with the platform and is influenced by many factors, mainly OS version/service pack and hardware configuration. Especially the hardware configuration can significantly enlarge the number of available return instructions since the installed drivers add a large codebase to the system: We found that often graphic card drivers add thousands of instructions that can be used by an attacker. For the complete codebase we found that on average every 162nd instruction is a return. Therefore an attacker typically finds tens of thousands of instructions she can use.

If the attacker restricts herself to using only the core kernel components, she is still able to find enough re-

turn instructions to be able to construct all necessary gadgets: We found that on average every 153rd instruction is a return, indicating a more dense structure within the core kernel components. These returns and the preceding instructions could be used to construct the gadgets in all tested environments. This result indicates that on Window-based systems an attacker can implement an arbitrary return-oriented program since all important gadgets can be built.

The most common instruction preceding a return is `pop ebp`: On average across all tested systems, this instruction was found in about 72% of the cases. This is no surprise since the sequence `pop ebp; ret` is the standard exit sequence for C code. Other common instructions the Constructor finds are `add esp, <const>` (12.2%), `pop (eax|ecx|edx)` (4.2%), and `xor eax, eax` (3.7%). Other instructions can be found rather seldom, but if a given instruction occurs at least once the attacker can use it. For example, the instruction `lahf`, which is used to access the CPU's flag registers, was commonly found less than 10 times, but nevertheless the attacker can take advantage of it.

4.1.2 Gadget Examples

In order to illustrate the gadgets constructed by our framework, we present a few examples of gadgets in this section. A full listing of all gadgets constructed during the evaluation on ten different machines is available online [13] such that our results can be verified.

Figure 5 shows the AND gadget constructed on two different machines both running Windows XP SP2. In each of the sub-figures, the left part displays the instructions that are actually used for the computation: Remember that our current implementation considers one instruction preceding a return instruction, i.e., after each of the displayed instructions one implicit `ret` instruction is executed. The right part shows the memory locations where the instruction is found within kernel memory (R), or indicates the label name (L). Labels are memory variable addresses.

The two gadgets each perform a logical AND of two values. This is achieved by loading the two operands into the appropriate registers (`pop, mov` sequence), then performing the `and` instruction on the registers, and finally writing the result back to the destination address. Although both programs are executed on Windows XP SP2 machines, the resulting return-oriented code looks completely different since useful instructions in different kernel components are used by the Constructor.

Another example of a gadget constructed by our framework is shown in Figure 6. The left example shows a gadget for a machine running Windows Vista, while the example on the right hand side is constructed on a ma-

Machine configuration	# ret inst.	# trie leaves	# ret inst. (res)	# trie leaves (res)
Native / XP SP2	118,154	148,916	22,398	25,968
Native / XP SP3	95,809	119,533	22,076	25,768
VMware / XP SP3	58,933	67,837	22,076	25,768
VMware / 2003 Server SP2	61,080	70,957	23,181	26,399
Native / Vista SP1	181,138	234,685	30,922	36,308
Bootcamp / Vista SP1	177,778	225,551	30,922	36,308

Table 1: Overview of return instructions found and generated trie leaves on different machines

pop ecx	R: ntkrnlpa.exe:0006373C	pop ecx	R: nv4_mini.sys:00005A15
mov edx, [ecx-0x4]	L: <RightSourceAddress>+4		L: <RightSourceAddress>-4
pop eax	R: ntkrnlpa.exe:000436AE	pop eax	R: nv4_mini.sys:00074EF2
	L: <LeftSourceAddress>		L: <LeftSourceAddress>
mov eax, [eax]	R: win32k.sys:000065D1	mov eax, [eax]	R: nv4_disp.dll:00125F30
and eax, edx	R: win32k.sys:000ADAE6	and eax, [ecx+0x4]	R: sthda.sys:000024ED
pop ecx	R: ntkrnlpa.exe:0006373C	pop ecx	R: nv4_mini.sys:00005A15
	L: <DestinationAddress>		L: <DestinationAddress>
mov [ecx], eax	R: win32k.sys:0000F0AC	mov [ecx], eax	R: nv4_disp.dll:000DE9DA

Figure 5: Example of two AND gadgets constructed on different machines running Windows XP SP2. The implicit ret instruction after each instruction is omitted for the sake of brevity.

chine running Windows 2003 Server. Again, the memory locations of the gadget instructions are completely different since the Constructor found different useful instruction sequences that are then used to build the gadget.

4.2 Runtime Overhead

The average runtime of the Constructor for the restricted set of drivers that should be analyzed is 2,009 ms, thus the time for finding and constructing the final gadgets is rather small.

To assess the overhead of return-oriented programming in real-world settings, we also measured the overhead of an example program written within our framework compared to a “native” implementation in C. Therefore, we implemented two identical versions of QuickSort, one in C and one in our dedicated return-oriented language. The source code of the latter can be seen in Appendix B.

Both algorithms sort an integer array of 500,000 randomly selected elements and the evaluations were carried out on an Intel Core 2 Duo T7500 based notebook running Windows XP SP3. The C code was compiled with Microsoft Visual Studio 2008; in order to improve the fundamental expressiveness of the comparison, all compiler optimizations were disabled. Each algorithm was executed three times and we calculated the average of the runtimes.

The return-oriented QuickSort took *21,752 ms* on average compared to *161 ms* for C QuickSort. The results clearly show that the overhead imposed by return-

oriented programs is significant; on average, they were 135 times slower than their C counterparts. We would like to stress that we did not build our system with speed optimizations in mind. Additionally, in our domain, return-oriented rootkits usually do not involve time-intensive computations, thus the slowness might not be a problem in practice. On the other hand, the overhead might well be exploited by detection mechanisms that try to find return-oriented programs.

5 Return-Oriented Rootkit

In order to evaluate our system in the presence of a kernel vulnerability, we have implemented a dedicated driver containing insecure code. Remember that our attack model includes this situation. By this example, we show that our systems allows us to implement a return-oriented rootkit in an efficient and portable manner. This rootkits bypasses kernel code integrity mechanisms like NICKLE and SecVisor since we do not inject new code into the kernel, but only execute code that is already available. While the authors of NICKLE and SecVisor acknowledge that such a vulnerability could exist [19, 22], we are the first to actually show an implementation of an attack against these systems. In the following, we first introduce the different stages of the infection process and afterwards describe the internals of our rootkit example.

'LoadEspPointer' gadget:		'LoadEspPointer' gadget:	
pop ecx	R: nvlddmkm.sys:000156F5	pop eax	R: ntkrnlpa.exe:0001CD4F
	L: <Address>		L: <Address>
mov eax, [ecx]	R: ntkrnlpa.exe:002D15C3	mov eax, [eax]	R: win32k.sys:00087E17
mov eax, [eax]	R: win32k.sys:000011AE	mov eax, [eax]	R: win32k.sys:00087E17
pop ecx	R: nvlddmkm.sys:000156F5	pop ecx	R: ntkrnlpa.exe:00080A8D
	L: &<LocalVar>		L: &<LocalVar>
mov [ecx], eax	R: ntkrnlpa.exe:0002039B	mov [ecx], eax	R: win32k.sys:000A8DDB
pop esp	R: nvlddmkm.sys:00036A54	pop esp	R: ntkrnlpa.exe:00081A67
	L: <LocalVar>		L: <LocalVar>

Figure 6: Example of gadget constructed on a machine running Windows Vista SP1 (left) and Windows 2003 Server (right). Again, the implicit `ret` instruction after each instruction is omitted.

5.1 Experimental Setup

Vulnerability. As already stated, we assume the presence of a vulnerability in kernel code that enables an exploit to corrupt the program flow in kernel mode. More precisely, our dedicated driver contains a specially crafted buffer overflow vulnerability that allows an attacker to tamper with the kernel stack. The usual way to implement driver-to-process communication is to provide a device file name being accessible from userspace. The process hence opens this device file and may send data to the driver by writing to it. Write requests trigger so-called *I/O request packets* (IRP) at the driver's callback routine. The driver then takes the input data from userspace and copies it into its own local buffer without validating its length. This leads to a classical buffer overflow attack and enables us to write stack values of arbitrary length.

Exploit. We exploit this vulnerability by writing an oversized buffer to the device file, thereby replacing the return value on the stack to point to a `pop esp; ret` sequence, and the next stack value to point to the entrypoint of the return-oriented program. By overwriting these eight bytes, we manage to modify the stack register to point to the beginning of our return-oriented program. Of course, the vulnerability itself may vary in its concrete nature, however, any similar insecure code allows us to mount our attack: A single vulnerability within the kernel or any third-party driver is enough to attack a system and start the return-oriented program.

The only question that remains is where to put the program image. We basically have two options: First, the exploit could overwrite the entire kernel stack with our return-oriented program; in case of the above vulnerability, this would be possible as there is no upper limit. In case of Windows, the kernel stack size has a fixed limit of 3 pages which heavily constrains this option. Second, the exploit could, at least initially, keep the program image in userspace memory. We prefer the latter approach

to implement our *rootkit loader*, although it has some implications that need to be addressed as we now explain.

5.2 Intricacies in Practice

One of the main practical obstacles that we faced stems from the way how Windows treats its kernel stack. All current Windows operating systems separate kernel space execution into several *interrupt request levels* (IRQL). IRQLs introduce a priority mechanism into kernel-level execution and are similar to user-level thread priorities. Every interrupt is executed in a well-defined IRQL; whenever such an interrupt occurs, it is compared to the IRQL of the currently executing thread. In case the current IRQL is above the requested one, the interrupt is queued for later rescheduling. As a consequence, an interrupt cannot suspend a computation running at a higher IRQL. This has some implications concerning accessing pageable memory in kernel mode since page-access exceptions are being processed in a specific IRQL (`APC_LEVEL`, to be precise) while other interrupts are handled at higher IRQLs. Hence, the kernel and drivers must not access pageable memory areas at certain IRQLs.

Unfortunately, this leads to some problems due to a peculiarity of Windows kernels: Whenever interrupts occur and hence must be handled, the Windows kernel borrows the current kernel stack to perform its interrupt handling. Therefore, the interrupt handler allocates the memory *below* the current value of `esp` as the handler's stack frame. While this is totally acceptable in common situations, it leads to undesirable implications in case of return-oriented programs as the stack values below the current stack pointer may indeed be needed in the subsequent execution. As described in Section 3.1.2, control flow branches are stack register modifications: When the program wants to jump backwards, it may fail at this point since the prior code might have been overwritten by the interrupt handler in the first place. To solve this problem, the Compiler provides an option to dynamically restore affected code areas: Whenever a return-oriented

control flow transition backwards occurs (which hence could have been subject to unsolicited modifications), we first prepare the ICA to perform a `memcpy` call that restores the affected code from the backup code section and subsequently performs the return-oriented jump. This works since the ICA is located below the code section and hence the code section cannot be overwritten during the call. The data and backup section will never be overwritten as they are always on top of every possible value of `esp`.

Furthermore, we will also run into IRQL problems in case the program stack is located in pageable memory: As soon as an interrupt is dispatched above `APC_LEVEL`, a blue-screen-of-death occurs. This problem should be overcome by means of the `VirtualLock` function which allows a process to lock a certain amount of pages into physical memory, thereby eliminating the possibility of paging errors on access. However, due to reasons which are yet not known to us, this does not always work as intended for memory areas larger than one page. We have frequently encountered paging errors in kernelmode although the related memory pages have previously been locked. We therefore introduce a workaround for this issue in the next section.

5.3 Rootkit Loader

To overcome the paging IRQL problem, we have implemented a pre-step in the loading phase. More precisely, in the first stage, we prepare a tiny return-oriented rootkit loader that fits into one memory page and prepares the entry of the actual return-oriented rootkit. It allocates memory from the kernel's non-paged pool, which is definitely never paged out, and copies the rootkit code from userspace before performing a transition to the actual rootkit. This has proven to work reliably in practice and we have not encountered any further IRQL problems. Again, the Rootkit Loader program image resides in userspace, which limits the ability of kernel integrity protection mechanisms to prohibit the loading of our rootkit.

5.4 Rootkit Implementation

To demonstrate our system's capability, we have implemented a return-oriented rootkit that is able to hide certain system processes. This is achieved by an approach similar to the one introduced by Høglund and Butler [11]: Our rootkit cycles through Windows' internal process block list to search for the process that should be hidden and, if successful, then modifies the pointers in the doubly-linked list accordingly to release the process' block from the list. Since the operating system holds a separate, independent scheduling list, the process will

```
int ListStartOffset =
    &CurrentProcess->process_list.Flink -
    CurrentProcess;
int ListStart =
    &CurrentProcess->process_list.Flink;
int ListCurrent = *ListStart;
while(ListCurrent != ListStart) {
    struct EPROCESS *NextProcess =
        ListCurrent - ListStartOffset;
    if(RtlCompareMemory(NextProcess->ImageName,
        "Ghost.exe", 9) == 9) {
        break;
    }
    ListCurrent = *ListCurrent;
}

if(ListCurrent != ListStart) {
    // process found, do some pointer magic
    struct EPROCESS *GhostProcess =
        ListCurrent - ListStartOffset;

    // Current->Blink->Flink = Current->Flink
    GhostProcess->process_list.Blink->Flink =
        GhostProcess->process_list.Flink;

    // Current->Flink->Blink = Current->Blink
    GhostProcess->process_list.Flink->Blink =
        GhostProcess->process_list.Blink;

    // Current->Flink = Current->Blink = Current
    GhostProcess->process_list.Flink =
        ListCurrent;
    GhostProcess->process_list.Blink =
        ListCurrent;
}
```

Figure 7: Rootkit source code snippet in dedicated language for return-oriented programming that can be compiled with our Compiler.

still be running in the system, albeit not being present in the results of process enumeration requests: The process is hidden within Windows and not visible within the Taskmanager. Figure 8 in Appendix A illustrates the rootkit in practice.

Figure 7 shows an excerpt of the rootkit source code written in our dedicated language. This snippet shows the code for (a) finding the process to be hidden and (b) hiding the process as explained above.

Once the process hiding is finished, the rootkit performs a transition back to the vulnerable code to continue normal execution. This seems to be complicated since we have modified the stack pointer in the first place and must hence restore its original value. However, in practice this turns out to be not problematic since this value is available in the thread environment block that is always located at a fixed memory location. Hence, we reconstruct the stack and jump back to our vulnerable driver. Besides process hiding, arbitrary data-driven attacks can be implemented in the same way: The rootkit needs to

exploit the vulnerability repeatedly in order to gain control and can then execute arbitrary return-oriented programs that perform the desired operation [3].

We would like to mention at this point that more sophisticated rootkit functionality, e.g., file and socket hiding, might demand more powerful constructs, namely *persistent* return-oriented callback routines. Data-only modifications as implemented by our current version of the rootkit hence might not be sufficient in this case. In contrast to Riley et al. [19], we do believe that this is possible in the given environment by the use of specific instruction sequences. However, we have not yet had the time to prove our hypothesis and hence leave this topic up to future work in this area.

The rootkit example works on Windows 2000, Windows Server 2003 and Windows XP (including all service packs). We did not port it to the Vista platform yet as the publicly available information on the Vista kernel is still limited. We also expect problems with the Vista *PatchGuard*, a kernel patch protection system developed by Microsoft to protect x64 editions of Windows against malicious patching of the kernel. However, we would like to stress that PatchGuard runs at the same privilege level as our rootkit and hence could be defeated. In the past, detailed reports showed how to circumvent Vista PatchGuard in different ways [24, 26, 25].

6 Conclusion and Future Work

In this paper we presented the design and implementation of a framework to automate return-oriented programming on commodity operating systems. This system is portable in the sense that the Constructor first enumerates what instruction sequences can be used and then dynamically generates gadgets that perform higher-level operations. The final gadgets are then used by the Compiler to translate the source code of our dedicated programming language into a return-oriented program. The language we implemented resembles the syntax of the C programming language which greatly simplifies developing programs within our framework. We confirmed the portability and universality of our framework by testing the framework on ten different machines, providing deeper insights into the mechanisms and constraints of return-oriented programming. Finally we demonstrated how a return-oriented rootkit can be implemented that circumvents kernel integrity protection systems like NICKLE and SecVisor.

In the future, we want to investigate effective detection techniques for return-oriented rootkits. We also plan to extend the research in two other important directions. First, we plan to examine how the current rootkit can be improved to also support *persistent* kernel modifications. This change is necessary to implement rootkit functions

like hiding of files or network connections, which require a persistent return-oriented callback routine. This change would enhance the rootkit beyond the current data-driven attacks. Second, we plan to analyze how the techniques presented in this paper could be used to attack control-flow integrity [1, 7, 14, 18] or data-flow integrity [5] mechanisms. These mechanisms are orthogonal to the kernel integrity protection mechanisms we covered in this paper.

References

- [1] Martin Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-Flow Integrity – Principles, Implementations, and Applications. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, November 2005.
- [2] James P. Anderson. Computer Security Technology Planning Study. Technical Report ESD-TR-73-51, AFSC, Hanscom AFB, Bedford, MA, October 1972. AD-758 206, ESD/AFSC.
- [3] Arati Baliga, Pandurang Kamat, and Liviu Iftode. Lurking in the Shadows: Identifying Systemic Threats to Kernel Data. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, 2007.
- [4] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)*, October 2008.
- [5] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.
- [6] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems. In *Proceedings of the 13th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, May 2008.
- [7] John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram Adve. Secure Virtual Architecture: A Safe Execution Environment for Commodity Operating Systems. *SIGOPS Oper. Syst. Rev.*, 41(6), 2007.

- [8] Jason Franklin, Arvind Seshadri, Ning Qu, Sagar Chaki, and Anupam Datta. Attacking, Repairing, and Verifying SecVisor: A Retrospective on the Security of a Hypervisor. Technical Report CyLab Technical Report CMU-CyLab-08-008, CMU, June 2008.
- [9] Tal Garfinkel and Mendel Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of the 10th Network and Distributed Systems Security Symposium (NDSS)*, February 2003.
- [10] Gil Dabah. diStorm64 - The ultimate disassembler library. <http://www.ragestorm.net/distorm>, 2009.
- [11] Greg Hoglund and Jamie Butler. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley Professional, July 2005.
- [12] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley, 2006.
- [13] Ralf Hund. Listing of gadgets constructed on ten evaluation machines. <http://pil.informatik.uni-mannheim.de/filepool/projects/return-oriented-rootkit/measurements-ro.tgz>, May 2009.
- [14] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. Secure Execution via Program Shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–206, 2002.
- [15] Microsoft. Digital Signatures for Kernel Modules on Systems Running Windows Vista. <http://download.microsoft.com/download/9/c/5/9c5b2167-8017-4bae-9fde-d599bac8184a/kmsigning.doc>, July 2007.
- [16] Microsoft. A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2. <http://support.microsoft.com/kb/875352>, 2008.
- [17] Nergal. The advanced return-into-lib(c) exploits: PaX case study. <http://www.phrack.org/issues.html?issue=58&id=4>, 2001.
- [18] Nick L. Petroni, Jr. and Michael Hicks. Automated Detection of Persistent Kernel Control-Flow Attacks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, pages 103–115, October 2007.
- [19] Ryan Riley, Xuxian Jiang, and Dongyan Xu. Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing. In *Proceedings of the 11th Symposium on Recent Advances in Intrusion Detection (RAID)*, 2008.
- [20] Ryan Riley, Xuxian Jiang, and Dongyan Xu. NICKLE: No Instructions Creeping into Kernel Level Executed. <http://friends.cs.purdue.edu/dokuwiki/doku.php?id=nickle>, 2008.
- [21] Sebastian Krahmer. x86-64 Buffer Overflow Exploits and the Borrowed Code Chunks Exploitation Techniques. <http://www.suse.de/~krahmer/no-nx.pdf>, September 2005.
- [22] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, 2007.
- [23] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, October 2007.
- [24] skape and Skywing. Bypassing PatchGuard on Windows x64. *Uninformed*, 3, January 2006.
- [25] Skywing. PatchGuard Reloaded: A Brief Analysis of PatchGuard 3. *Uninformed*, 8, September 2007.
- [26] Skywing. Subverting PatchGuard Version 2. *Uninformed*, 6, January 2007.
- [27] Solar Designer. Getting around non-executable stack (and fix). <http://seclists.org/bugtraq/1997/Aug/0063.html>, 1997.
- [28] PaX Team. Documentation for the PaX project - overall description. <http://pax.grsecurity.net/docs/pax.txt>, 2008.
- [29] Terence Parr. ANTLR Parser Generator. <http://www.antlr.org>, 2009.
- [30] A. M. Turing. On Computable Numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc.*, 2(42):230–265, 1936.

A Return-Oriented Rootkit in Practice

Figure 8 depicts the results of an attack using our return-oriented rootkit: The process Ghost.exe (lower left window) is a simple application that periodically prints a status message on the screen. The rootkit (upper left window) first exploits the vulnerability in the driver to start

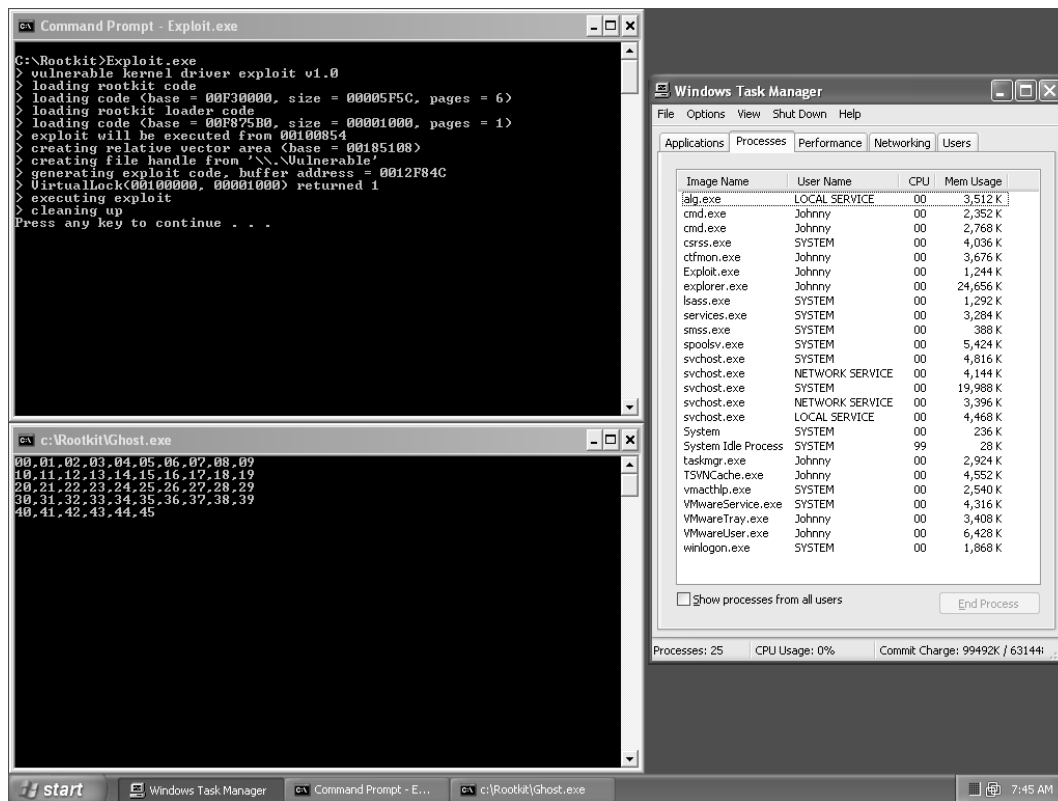


Figure 8: Return-oriented rootkit in practice, hiding the process Ghost.exe.

the return-oriented program. This program then hides the presence of Ghost.exe as explained in Section 5.4: The process Ghost.exe is running, however, the rootkit removed it from the list of running processes and it is not visible in the Taskmanager.

B Return-Oriented QuickSort

The following listing shows an implementation of QuickSort within our dedicated programming language. The syntax is close to the C programming language, allowing a programmer to implement a return-oriented program without too much effort. The most notable exception from C's syntax is related to importing of external functions: Our language can import subroutine calls from other libraries, enabling an easy way to call external functions like `printf()` or `srand()`. However, each function needs to be imported explicitly. Furthermore, the language implements only a basic type-system consisting of integers and character arrays, but this should not pose a limitation.

```
import("msvcrt.dll", printf:cdecl,
      srand:cdecl,
      rand:cdecl,
      malloc:cdecl);
```

```
import("kernel32.dll", GetCurrentProcess,
      TerminateProcess,
      GetTickCount);
```

```
int data;
int size = 500000;
```

```
function partition(int left, int right,
                  int pivot_index) {
    int pivot = data[pivot_index];
    int temp = data[pivot_index];
    data[pivot_index] = data[right];
    data[right] = temp;
    int store_index = left;
    int i = left;

    while(i < right) {
        if(data[i] <= pivot) {
            temp = data[i];
            data[i] = data[store_index];
            data[store_index] = temp;
            store_index = store_index + 1;
        }
        i = i + 1;
    }
}
```

```
temp = data[store_index];
data[store_index] = data[right];
data[right] = temp;
```

```
return store_index;
}
```

```

function quicksort(int left, int right) {
    if(left < right) {
        int pivot_index = left;
        pivot_index = partition(left, right,
                                pivot_index);
        quicksort(left, pivot_index - 1);
        quicksort(pivot_index + 1, right);
    }
}

function start() {
    printf("Welcome to ro-QuickSort\n");
    data = malloc(4 * size);
    srand(GetTickCount());
    int i = 0;
    while(i < size) {
        data[i] = rand();
        i = i + 1;
    }

    int time_start = GetTickCount();
    quicksort(0, size - 1);
    int time_end = GetTickCount();
    printf("Sorting completed in %u ms:\n",
           time_end - time_start);
}

```

Crying Wolf: An Empirical Study of SSL Warning Effectiveness

Joshua Sunshine, Serge Egelman, Hazim Almuhiemedi, Neha Atri, and Lorrie Faith Cranor
Carnegie Mellon University
{sunshine, egelman, hazim}@cs.cmu.edu, natri@andrew.cmu.edu, lorrie@cs.cmu.edu

Abstract

Web users are shown an invalid certificate warning when their browser cannot validate the identity of the websites they are visiting. While these warnings often appear in benign situations, they can also signal a man-in-the-middle attack. We conducted a survey of over 400 Internet users to examine their reactions to and understanding of current SSL warnings. We then designed two new warnings using warnings science principles and lessons learned from the survey. We evaluated warnings used in three popular web browsers and our two warnings in a 100-participant, between-subjects laboratory study. Our warnings performed significantly better than existing warnings, but far too many participants exhibited dangerous behavior in all warning conditions. Our results suggest that, while warnings can be improved, a better approach may be to minimize the use of SSL warnings altogether by blocking users from making unsafe connections and eliminating warnings in benign situations.

1 Introduction

Browsers display Secure Socket Layer (SSL)¹ warnings to warn users about a variety of certificate problems, for example when the server's certificate has expired, mismatches the address of the server, or is

signed by an unrecognized authority. These warning messages sometimes indicate a man-in-the-middle or DNS spoofing attack. However, much more frequently users are actually connecting to a legitimate website with an erroneous or self-signed certificate.

The warnings science literature suggests that warnings should be used only as a last resort when it is not possible to eliminate or guard against a hazard. When warnings are used, it is important that they communicate clearly about the risk and provide straightforward instructions for avoiding the hazard [19, 22]. In this paper we examine user reactions to five different SSL warnings embodying three strategies: make it difficult for users to override the warning, clearly explain the potential danger facing users, and ask a question users can answer. By making it difficult for users to override the warning and proceed to a potentially dangerous website, the warning may effectively act as a guard against the hazard, similarly to the way a fence protects people from falling into a hole. While some people may still climb the fence, this requires extra effort. By clearly explaining the potential danger, warnings communicate about risk. Finally, by asking users a question they can answer, the system can tailor a warning to the user's situation and instruct users in the appropriate steps necessary to avoid any hazard.

We conducted a survey of 409 Internet users' reactions to current web browser SSL warnings and found that risk perceptions were the leading factor in respondents' decisions of whether or not to visit a website with an SSL error. However, those who understood the risks also perceived some common SSL warnings as not very risky, and were more likely to override those warnings.

¹The Secure Socket Layer (SSL) and Transport Layer Security (TLS) protocols secure web communication by encrypting data sent between browser and server and by validating the identity of the server. For the remainder of the paper we will use the common convention of using the term "SSL" to refer to both protocols.

We followed up this survey with a between-subjects laboratory experiment involving 100 participants who encountered SSL warnings on an online banking website that requested their credentials and a library website that did not request any credentials. We tested the Firefox 2 (FF2), Firefox 3 (FF3), and Microsoft Internet Explorer 7 (IE7) SSL warnings. We also tested two new warnings designed to take advantage of the lessons we learned in the survey. The first warning was designed with risk in mind: it succinctly explained the risks and consequences of proceeding to the website. The second warning was context sensitive: it appeared to be more severe when the participants visited websites that required them to enter personal data. We found that most participants ignored the FF2 and IE7 warnings on both websites. Many participants who used FF3 were unable to override that warning and were thus prevented from visiting both websites. Finally, we found that participants who viewed our redesigned warnings better understood the risks and made their decisions based on the type of website they were visiting. However, despite the fact that the warnings we examined embodied the best techniques available, none of the warnings provided adequate protection against man-in-the-middle attacks. Our results suggest that, while warnings can be improved, a better approach may be to minimize the use of SSL warnings altogether by blocking users from making unsafe connections and eliminating warnings in benign situations.

In the next section we provide an overview of other studies that have been conducted on web browser security indicators. In Section 3 we present our online SSL warning survey methodology and results. In Section 4 we present our laboratory experiment methodology and results. Finally, we discuss our findings and conclusions.

2 Background and Related Work

Much previous research has indicated that users do not understand SSL. A study in 2002 found that half of the participants could not identify a secure browser

connection [8]. A 2005 study tracked eye movements and found that participants paid no attention to web browser security cues such as SSL icons. Only after priming participants to be on the lookout for security information, 69% of participants noticed the lock icon [21]. Schechter et al. tested the usability of security indicators by removing SSL indicators from a banking website and observed that all 63 participants still provided their passwords [17].

The major web browsers now include support for extended validation (EV) certificates. A regular certificate only tells a user that the certificate was granted by a particular issuing authority, whereas an EV certificate also says that it belongs to a legally recognized corporate entity [2]. FF3 and IE7 indicate a website has an EV certificate by coloring the address bar green and displaying the name of the website owner. However, a study by Jackson et al. found that EV certificates did not make users less likely to fall for phishing attacks. Many users were confused when the chrome of the web browser was spoofed within the content window to depict a green address bar. Additionally, after reading a help file, users were less suspicious of fraudulent websites that did not yield warning indicators [11]. Sobey et al. performed an eye tracking study in 2008 to examine whether participants would notice simulated versions of the EV certificate indicators that are used by FF3. They found that none of their 28 participants examined the address bar when making online shopping decisions, and therefore none of them encountered the secondary SSL dialogs containing information about the website owners [18].

Usability problems with security indicators in web browsers go beyond SSL. Wu et al. conducted a study of security toolbars used to help users identify phishing websites. The researchers examined three different styles of passive indicators—indicators that do not force user interactions—that appeared in the browser chrome. They discovered that 25% of the participants failed to notice the security indicators because they were focused on the primary task. In fact, many of those who did notice the indicators did not trust them because they believed the tool was in error since the website looked trustworthy [23]. The factors that go into website trust have been exten-

sively studied by Fogg et al., who found that the “look and feel” of a website is often most important for gaining user trust [7]. Thus users might trust a professional looking website despite the presence of a passive security indicator. Dhamija et al. corroborated these findings by performing a study on why people fall for phishing websites. In their study, users examined a set of websites and were asked to identify which ones were phishing websites. They found that 23% of their study participants did not look at any of the web browser security indicators when making their decisions, even though the participants were primed for security. The researchers concluded that passive security indicators are ineffective because they often go unnoticed [4].

Because of the problems with passive security indicators, many web browsers now display “active” warnings that require the user to take an action—usually deciding whether or not to visit the destination website—in order to dismiss the warning. While these warnings force the user to acknowledge them, they still allow the user to ignore their advice and proceed to the website despite the security error. In 2008, Egelman et al. performed a study on active web browser warnings used to warn users about potential phishing websites. They discovered that users who claimed to have seen the warnings before were significantly more likely to ignore them in the laboratory. They concluded that many of the participants had become habituated to seeing similar-looking warnings when browsing legitimate websites, and are now likely to ignore all future similarly-designed warnings, regardless of the danger they represent [6].

Jackson and Barth address the problem of users ignoring SSL warnings with the ForceHTTPS system [10]. Websites with CA signed certificates deploy a special ForceHTTPS cookie to a user’s browser, which from then on only accepts valid SSL connections to the website. This strategy is elegantly simple, but it does not protect users when they encounter a website for the first time.

Wendlandt et al. created the Perspectives system to prevent habituation by only displaying warnings when an attack is probable. Perspectives transforms the CA model into a “trust-on-first-use” model, similar to how SSH works. “Notaries” keep track

of all previously viewed SSL certificates and only warn users when they encounter a certificate that has changed over time. This eliminates many common SSL errors, thereby only displaying warnings when an attack is probable [20]. However, when users do encounter certificates that have been altered, it is unclear how the warnings should be designed so as to maximize their effectiveness.

Xia and Brustoloni implement a system to help users better react to unverified certificates [24]. The system requires websites interested in using private CA signed certificates to distribute tokens to their users by physical media. In 2007, Brustoloni and Villamarín-Salomón explored the idea of creating polymorphic dialogs to combat habituation. While their preliminary results were promising for warning users about malicious email attachments, it is unclear what the long-term efficacy would be if such a system were created for SSL warnings [1].

The pervasive nature of SSL errors raises questions about the efficacy of SSL warnings. A survey of 297,574 SSL-enabled websites queried in January 2007 found 62% of the websites had certificates that would trigger browser warnings [5]. A January 2009 study performed using a list of the top one million websites found that at least 44% of the 382,860 SSL-enabled websites had certificates that would trigger warnings [13].² Given this large sample, many of the errors may appear on websites that are not frequently visited. Our own analysis of the top 1,000 SSL-enabled websites yielded 194 SSL errors, which is still an alarming number. Unfortunately, we do not have data on the proportion of certificate errors that appear on legitimate websites versus malicious websites, making it unclear whether these particular errors are indicative of an ongoing attack. However, we believe it is likely that most certificate errors occur on non-malicious websites, and therefore many users view the associated warnings as false positives. This means that if a web browser displays a particular warning each time it encounters any type of certificate error, users will quickly become habituated to this warning regardless of the underlying error.

²This estimate is likely low as the 2009 study did not catalog domain name mismatch errors.

3 SSL Survey

In the summer of 2008 we conducted an online survey of Internet users from around the world to determine how they perceived the current web browser SSL warnings.

3.1 Methodology

We presented survey respondents with screenshots of three different SSL warnings from the browser that they were using at the time they took the survey³ and asked them several questions about each warning. These questions were followed by a series of questions to determine demographic information.

We showed participants warnings for expired certificates, certificates with an unknown issuer, and certificates with mismatched domain names.⁴ Each warning was shown on a separate page along with its associated questions, and the order of the three pages was randomized. We included a between-group condition to see if context played a role in users' responses: half the participants were shown a location bar for *craigslist.org*—an anonymous forum unlikely to collect personal information—and the other half were shown a location bar for *amazon.com*—a large online retailer likely to collect personal and financial information. We hypothesized that respondents might be more apprehensive about ignoring the warning on a website that was likely to collect personal information. Below each warning screenshot, participants were asked a series of questions to determine whether they understood what the warnings mean, what they would do when confronted with each warning, and their beliefs about the consequences of ignoring these warnings.

We were also interested in determining how computer security experts would respond to our survey, and if the experts' answers would differ from everyone else's answers. In order to qualify respondents as experts, we asked them a series of five ques-

tions to determine whether they had a degree in an IT-related field, computer security job experience or course work, knowledge of a programming language, and whether they had attended a computer security conference in the past two years.

We recruited participants from Craigslist and several contest-related bulletin boards, offering a gift certificate drawing as an incentive to complete the survey. We received 615 responses; however we used data from only the 409 respondents who were using one of the three web browsers under study.

3.2 Analysis

Our 409 survey respondents used the following browsers: 96 (23%) used FF2, 117 (29%) used FF3, and 196 (48%) used IE7. While age and gender were not significant predictors of responses,⁵ it should be noted that 66% of our respondents were female, significantly more males used FF3 ($\chi^2_2 = 34.01$, $p < 0.0005$), and that IE7 users were significantly older ($F_{2,405} = 19.694$, $p < 0.0005$). For these reasons and because respondents self-selected their web browsers, we analyzed the responses for each of the web browsers separately.

We found no significant differences in responses based on the type of website being visited. We found that respondents' abilities to correctly explain each warning was a predictor of behavior, though not in the way we expected: respondents who understood the domain mismatch warnings were less likely to proceed whereas we observed the opposite effect for the expired certificate warnings. This suggests that participants who understood the warnings viewed the expired certificate warnings as low risk. Finally, we found that risk perceptions were a leading factor in respondents' decisions and that many respondents—regardless of expertise—did not understand the current warnings. In this section we provide a detailed analysis of our results in terms of warning comprehension and risk perceptions, the role of context, and the role of expertise.

³We used screenshots of the warnings from FF2, FF3, and IE7. Users of web browsers other than FF2, FF3, or IE7 were only asked the demographic questions.

⁴We examined these three warnings in particular because we believed them to be the most common.

⁵All statistics were evaluated with $\alpha=0.05$. We used a Fisher's exact test for all statistics where we report a p-value only.

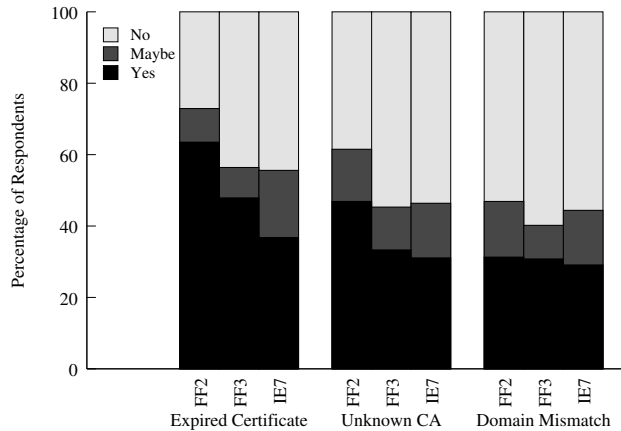


Figure 1: Participant responses to the question: *If you saw this message, would you attempt to continue to the website?*

3.2.1 Comprehension and Risk Perceptions

We were primarily interested in whether respondents would continue to the destination website if they saw a given warning. As shown in Figure 1, less than half the participants claimed they would continue.

We expected to see differences in behavior for each of the three types of warnings. In order for this to be the case, participants needed to be able to distinguish each of the three warnings. We asked them to explain what they thought each warning meant and coded the answers in terms of whether or not they were correct. As shown in Table 1, we discovered that FF2 users were significantly more likely to understand the domain mismatch warnings, while FF3 users were significantly more likely to understand the expired certificate warnings.

We explored warning comprehension further by examining whether those who understood the meaning of the warnings were more likely to heed or ignore them. In general, we found that users who understood the warnings tended to behave differently than those who did not. Across all three browsers, users who understood the domain mismatch warning were more likely to say they would heed that warning than users who did not understand it. In addition, FF3 and IE7 users who understood the expired certifi-

cate warnings were more likely to indicate that they would ignore these warnings and proceed to the destination website. These results are detailed in Table 1 and indicate that users likely perceive less risk when encountering an expired certificate, and therefore are likely to proceed. However, when encountering a domain mismatch warning, knowledgeable users perceive greater risk and are likely to discontinue.

The three warnings that we examined are displayed when the authenticity of the destination website’s SSL certificate cannot be guaranteed. While each of these warnings represents a different underlying error, they represent the same threat: the user may not be communicating with the intended website or a third party may be able to eavesdrop on her traffic. In both cases, sensitive information may be at risk (e.g. billing information when performing an online purchase). In order to determine whether or not respondents understood the threat model, we asked them to list the possible consequences of ignoring each of the warnings. Responses that specifically mentioned fraud, identity theft, stolen credentials (or other personal information), phishing, or eavesdropping were coded as being correct. We coded as correct 39% of responses for FF2 warnings, 44% of responses for FF3 warnings, and 37% of responses for IE7 warnings.

Incorrect responses fell into two categories: respondents who had no idea (or said there were no consequences) and respondents who mentioned other security threats. Many of those in the latter category mentioned viruses and worms. While it is possible that a malicious website may exploit web browser vulnerabilities or trick visitors into downloading malware, we considered these outside the scope of our survey because they either impact only users of a specific browser version—in the case of a vulnerability—or they rely on the user taking additional actions—such as downloading and executing a file. Several responses mentioned malware but additionally claimed that those using up-to-date security software are not at risk. Others claimed they were not at risk due to their operating systems:

“I use a Mac so nothing bad would happen.”
 “Since I use FreeBSD, rather than Windows, not much [risk].”

Browser	Understood	Expired Certificate			Unknown CA			Domain Mismatch		
				Ignored			Ignored			Ignored
FF2	Y	48	50%	71%	37	39%	43%	57	59%	19% $\chi^2_2 = 9.40$
	N	48	50%	56%	59	61%	49%	39	41%	49% $p < 0.009$
FF3	Y	55	47%	64% $\chi^2_2 = 21.05$	35	30%	31%	46	39%	15% $\chi^2_2 = 8.65$
	N	62	53%	34% $p < 0.0005$	82	70%	34%	71	61%	41% $p < 0.013$
IE7	Y	45	23%	53% $\chi^2_2 = 11.81$	44	22%	27%	62	32%	16% $\chi^2_2 = 7.50$
	N	151	77%	32% $p < 0.003$	152	78%	32%	134	68%	35% $p < 0.024$

Table 1: Participants from each condition who could correctly identify each warning, and of those, how many said they would continue to the website. Differences in comprehension within each browser condition were statistically significant (FF2: $Q_2 = 10.945$, $p < 0.004$; FF3: $Q_2 = 11.358$, $p < 0.003$; IE7: $Q_2 = 9.903$, $p < 0.007$). For each browser condition, the first line depicts the respondents who could correctly define the warnings, while the second depicts those who could not. There were no statistically significant differences between correctly understanding the unknown CA warning and whether they chose to ignore it.

“On my Linux box, nothing significantly bad would happen.”

Of course, operating systems or the use of security software do not prevent a user from submitting form data to a fraudulent website, nor do they prevent eavesdropping. We further examined risk perceptions by asking participants to specify the likelihood of “something bad happening” when ignoring each of the three warnings, using a 5-point Likert scale ranging from “0% chance” to “100% chance.” We found significant differences in responses to each warning for all three web browsers: respondents consistently ranked the expired certificate warning as being less risky than both of the other warnings. Table 2 depicts the perceived likelihood of risk for each of the web browsers and each of the three SSL warnings.

To examine whether there were differences in risk perception based on the underlying SSL error, we asked respondents to quantify the severity of the consequences of ignoring each of the SSL warnings using a 5-point Likert scale that ranged from “none” to “moderate” to “severe.” As shown in Table 3, we found that respondents in every web browser condition were likely to assign significantly lesser consequences to ignoring the expired certificate warning than when ignoring either of the other two warnings.

3.2.2 The Role of Expertise

Finally, we wanted to examine whether respondents’ level of technical expertise influenced their decisions to heed or ignore the warnings. As described in Section 3.1, we asked respondents a series of five questions to gauge their technical qualifications. We assigned each respondent a “tech score” corresponding to the number of questions they answered affirmatively. The first column of Table 4 lists the average scores for each of the web browser conditions. We classified those with tech scores greater than or equal to two as “experts.” The expert group represented the top 16.7% of FF2 users, the top 26.5% of FF3 users, and the top 12.2% of IE7 users. We compared our “experts” to the rest of our sample (i.e. respondents with scores of zero or one) and found that responses did not significantly differ in most cases. We found significant differences only among FF3 users when viewing the unknown CA and domain mismatch warnings: experts were significantly less likely to proceed to the websites (Table 4).

Finally, we examined whether the experts were better able to identify the individual warnings than the rest of the sample. We found that while the experts were more likely to identify the warnings than non-

	Expired Certificate	Unknown CA	Domain Mismatch	
FF2	37%	45%	54%	$\chi^2_2 = 25.19$ $p < 0.0005$
FF3	42%	52%	50%	$\chi^2_2 = 13.47$ $p < 0.001$
IE7	47%	52%	53%	$\chi^2_2 = 12.79$ $p < 0.002$

Table 2: Mean perceptions of the likelihood of “something bad happening” when ignoring each warning, using a 5-point Likert scale ranging from 0 to 100% chance. A Friedman test yielded significant differences for each browser.

	Expired Certificate	Unknown CA	Domain Mismatch	
FF2	1.70	2.10	2.29	$\chi^2_2 = 20.49$ $p < 0.0005$
FF3	1.96	2.36	2.32	$\chi^2_2 = 9.00$ $p < 0.011$
IE7	2.14	2.36	2.34	$\chi^2_2 = 16.90$ $p < 0.0005$

Table 3: Mean perceptions of the consequences of ignoring each of the three warnings, using a 5-point Likert scale ranging from 0 to 4. A Friedman test shows that respondents in every web browser condition were likely to assign significantly lesser consequences to ignoring the expired certificate warning than when ignoring either of the other two warnings.

experts, even in the best case, the experts were only able to correctly define the expired certificate warnings an average of 52% of the time, the unknown CA warnings 55% of the time, and the domain mismatch warnings 56% of the time. This indicates that either our metric for expertise needs to be improved, or that regardless of technical skills, many people are unable to distinguish between the various SSL warnings.

3.2.3 Conclusion

Our survey showed how risk perceptions are correlated with decisions to obey or ignore security warnings and demonstrated that those who understand security warnings perceive different levels of risk associated with each warning. However, a limitation of surveys is they collect participants’ self-reported data about what they think they would do in a hypothetical situation. Thus, it is useful to validate survey findings with experimental data.

4 Laboratory Experiment

We conducted a laboratory study to determine the effect of SSL warnings on user behavior during real tasks.

4.1 Methodology

We designed our laboratory study as a between-subjects experiment with five conditions: FF2 (Figure 2(a)), FF3 (Figure 3), IE7 (Figure 2(b)), a single-page redesigned warning (Figure 4(b)), and a multi-page redesigned warning (Figure 4). We asked participants to find information using four different types of information sources. Each task included a primary information source—a website—and an alternate source that was either an alternative website or a phone number. The primary information source for two of the tasks, the Carnegie Mellon University (CMU) online library catalog and an online banking application, were secured by SSL. We removed the certificate authorities verifying these websites from the trusted authorities list in each browser used in the study.⁶ Therefore, participants were shown an invalid certificate warning when they navigated to the library and bank websites. We noted how users reacted to these warnings and whether they completed the task by continuing to use the website or by switching to

⁶Ideally we would have performed a man-in-the-middle attack, for example by using a web proxy to remove the websites’ legitimate certificates before they reached the browser. However, due to legal concerns, we instead simulated a man-in-the-middle attack by removing the root certificates from the web browser.

Tech score			Expired	Unknown CA	Domain Mismatch	
FF2	$\mu = 0.61$	<i>Experts</i>	69%	44%	31%	
	$\sigma = 1.14$	<i>Non-Experts</i>	63%	48%	31%	
FF3	$\mu = 0.99$	<i>Experts</i>	52%	13%	$\chi^2_2 = 12.37$	$\chi^2_2 = 11.42$
	$\sigma = 1.42$	<i>Non-Experts</i>	47%	41%	$p < 0.002$	$p < 0.003$
IE7	$\mu = 0.47$	<i>Experts</i>	42%	33%	29%	
	$\sigma = 1.02$	<i>Non-Experts</i>	36%	31%	29%	

Table 4: Percentage of experts and non-experts who said they would continue past the warnings. The first column shows respondents’ average tech scores.

the alternative information source. Finally, we gave users an exit survey to gauge their understanding of and reaction to the warnings.

4.1.1 Recruitment

We recruited participants by posting our study on the experiment list of the Center for Behavioral Research at CMU. We also hung posters around the CMU campus. Participants were paid \$10–20 for their participation.⁷ All recruits were given an online screening survey, and only online banking customers of our chosen bank were allowed to participate. The survey included a range of demographic questions and questions about general Internet use.

In total, 261 users completed our screening survey and 100 users qualified and showed up to participate in our study. We randomly assigned 20 users to each condition. Half the users in each condition were given the bank task first and half were given the library task first. Participants took 15–35 minutes to complete the study including the exit survey.

We tried to ensure that participants were not primed to think about security. The study was presented not as a security study, but as a “usability of information sources study.” Our recruitment postings solicited people who were “CMU faculty staff or students” and had “used online banking in the last year.” However, we also required that participants have “purchased an item online in the last year” and “used a search engine” to avoid focusing potential participants on the banking tasks. Finally, our screening survey asked a series of questions whose

responses were not used to screen participants (e.g. “How often do you use Amazon.com?”), to further obfuscate the study purpose.

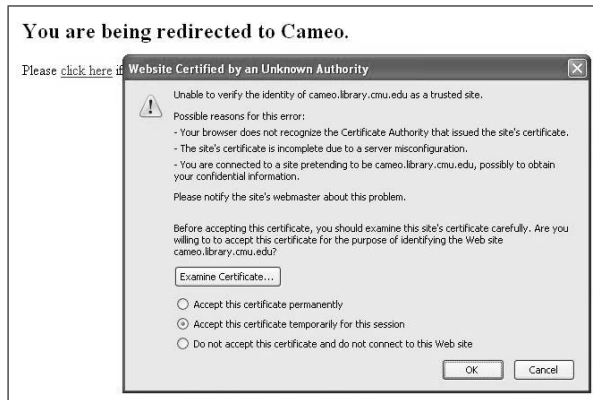
4.1.2 Conditions

The FF2 warning, displayed in Figure 2(a), is typical of invalid certificate warnings prior to 2006. This warning has a number of design flaws. The text contains jargon such as, “the website’s certificate is incomplete due to a server misconfiguration.” The look and feel of the warning, a grey dialog box with a set of radio buttons, is similar to a lot of other trivial dialogs that users typically ignore, such as “you are sending information unencrypted over the internet.” The default selection is to accept the certificate temporarily. This is an unsafe default for many websites, including the online banking application in our study.

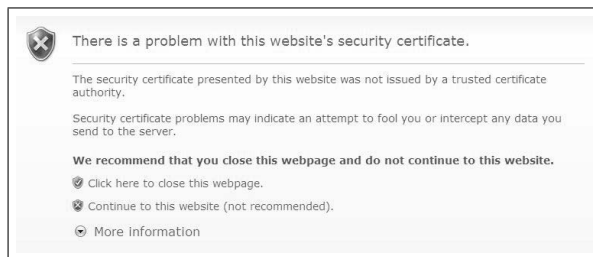
A more subtle problem with the FF2 warning, and those like it, is that it asks users a question that they cannot answer. The warning asks the user to determine if the certificate problem is the result of a server/browser configuration problem or a legitimate security concern. Since users are not capable of making this determination, the dialog is, in the words of Firefox project co-founder Blake Ross, “a dilemma to users.” Ross calls on browser designers to do everything possible to make decisions for their users. When designers have to ask questions of their users, they should ask questions that users can answer [16].

The FF3 warning should be more noticeable to users than its predecessor because it takes over the entire page and forces users to make a decision. Additionally, it takes four steps to navigate past the warning to the page with the invalid certificate. First

⁷Initially participants were paid \$10, but we raised the payment to \$20 to reach our recruiting goals.



(a) Firefox 2



(b) Internet Explorer 7

Figure 2: Screenshots of the FF2 and IE7 warnings.

the user has to click a link, mysteriously labeled “or you can add an exception. . .” (Figure 3), then click a button that opens a dialog requiring two more button clicks. The first version of the FF3 warning required 11 steps.⁸ This clearly represented a decision by Firefox developers that all invalid certificates are unsafe. They made the original version of the warning so difficult for users to override, that only an expert would be likely to figure out how to do it. While FF3 was in alpha and beta testing, many users erroneously believed the browser was in error when they could not visit websites that they believed to be legitimate.⁹

The IE7 warning, shown in Figure 2(b), occupies the middle ground between the FF2 and FF3 warnings. It takes over the entire page and has no default option, but differs from the FF3 warning because it

⁸https://bugzilla.mozilla.org/show_bug.cgi?id=399275

⁹https://bugzilla.mozilla.org/show_bug.cgi?id=398915



Figure 3: Screenshot of the initial FF3 warning.

can be overridden with a single click on a link labeled “Continue to this website.” It has a slightly scarier look and feel than the FF2 warning: the background color has a red tint and a large X in a red shield dominates the page. The warning also explicitly recommends against continuing. Finally, when viewing this warning the background of the address bar is red and continues to be red after one overrides the warning.

We designed two warnings using techniques from the warning literature and guided by results from our survey. Our multi-page warning first asks the user a question, displayed in Figure 4(a), and then, depending on the response, delivers the user either to the severe warning page shown in Figure 4(b) or to the requested website. The second version of the warning shows only the severe warning (Figure 4(b)). Both versions were implemented in IE7. We used the *resourcemodify* tool¹⁰ to replace the HTML file of the native warning in an IE DLL with our HTML files.

The second version of our warning serves two purposes. First, it attempts to see how users react to a simple, clear, but scary warning. The warning borrows its look and feel from the FF3 phishing warning. It is red and contains the most severe version of Larry the Firefox “passport officer.”¹¹ The title of the page is clear and harsh: “High Risk of Security Compromise.” The other context is similarly blunt (e.g. “an attacker is attempting to steal information that you are sending to *domain name*.”). Even the

¹⁰<http://deletethis.net/dave/xml-source-view/httperror.html>

¹¹http://news.cnet.com/8301-10789_3-9970606-57.html



(a) Page 1



(b) Page 2

Figure 4: Screenshot of redesigned warning.

default button, labeled “Get me out of here!” signifies danger. The only way for a user to continue is to click the tiny link labeled “Ignore this warning” in the bottom right corner. The second purpose of the single page warning is to help us interpret the results from our multi-page warning. We compare the multi-page results to the single-page results to see how the question affects user actions independent of the the scary second page.

The original FF3 warning aimed to avoid asking users questions, and instead decided on users’ behalf that invalid certificates are unsafe. However, even the Firefox designers eventually realized this could not work in the real world because too many legitimate websites use invalid certificates. Instead, our warning aims to ask the users a question that they can answer and will allow us to assess the risk level. Our question is, “What type of website are you trying to reach?” Users were required to select from one of four responses: “bank or other financial institution,” “online store or other e-commerce website,” “other,”

and “I don’t know.” If users selected the first two options, they saw the severe warning that discouraged them from continuing. We tested this question as a prototype for leveraging user-provided information to improve security warnings. It is not a complete solution as our question neglects many other types of websites that may collect sensitive information. We decided to show the secondary warning on bank websites and online stores because these are the most frequently attacked websites [15].

4.1.3 Experimental Setup

All studies were conducted in our laboratory on the same model of laptop. Participants interacted with the laptop within a virtual machine (VM). We reset the VM to a snapshot after each participant finished the study to destroy any sensitive data entered by the participant (e.g. bank password). This process also ensured that all browser and operating system settings were exactly the same for every participant. Finally, experimenters read instructions to participants from a script and experimenters did not help participants complete the tasks.

4.1.4 Tasks

After participants signed IRB consent forms, the experimenter handed them an instruction sheet and read this sheet aloud. Participants were reminded that they would be “visiting real websites and calling real organizations” and therefore should go about “each task in the way you would if you were completing it with the computer you usually use.” Participants were also instructed to “think aloud and tell us what you are thinking and doing as you complete each task,” in order to give us qualitative reactions to the warnings. The experimenter took notes throughout the study. The study was recorded (audio only), which allowed experimenters to retrieve details that were missed during note taking.

After the instructions were read and digested, the instruction sheets for each task were handed to the participant and read aloud by the experimenter one by one. The next task was not revealed until all previous tasks had been completed. The first task asked

participants to find the total area of Italy in square kilometers using Google or Ask.com as an alternative. The second task was to look up the last two digits of the participant's bank account balance using the online banking application or using phone banking. The third task was to locate the price of the hardcover edition of the book *Freakonomics* using Amazon.com or the Barnes and Noble website. Finally, the fourth task was to use the CMU online library catalog or alternatively the library phone number to retrieve the call number of the book *Richistan* (i.e. no personal information was transmitted).

The first and third tasks were “dummy tasks,” since the bookstore and search engine revealed no warnings. Instead, they reinforced to participants that the goal of the study was information sources, not security. Half the participants in each condition had the second and fourth tasks—the warning tasks—swapped so that we could control for the ordering of the warnings.

Researchers have found that study participants are highly motivated to complete assigned tasks. Participants want to please the experimenter and do not want to “fail” so they sometimes exert extreme effort to complete the task [12]. A closely related study [17] was criticized for not taking into account this “task focus” phenomenon [14]. Critics worried that participants were ignoring the warnings in the study because of task focus and not because this is what they would do in a more natural environment.

Our study design mitigates participants' task focus by presenting an alternate method for each task so that participants could “pass the test” without ignoring the warnings. We instructed participants to “try the suggested information source first,” to ensure that participants would only call the library or bank as a reaction to the warning. As there were no obstacles to completing the dummy tasks using the suggested information source, none of the participants used the alternate method to perform the dummy tasks.

4.1.5 Exit Survey

After completing all four study tasks, participants were directed to an online exit survey hosted by Sur-

veyMonkey. The exit survey asked 45 questions in six categories. The first set of questions asked about their understanding of and reaction to the bank warning in the study. The second question asked the same questions about the library warning. The third set asked questions to gauge their general understanding of certificates and invalid certificate warnings. The fourth set gauged participants' prior exposure to identity theft and other cyberthreats. The fifth set, which were also asked in the online SSL survey, asked them about their technical experience, including their experience with computer security. Finally, the sixth set asked general demographic questions like age, gender and education level.

4.2 Results and Analysis

The primary goal of any SSL warning should be to prevent users from transmitting sensitive information to suspicious websites. A secondary—but still important—goal is to allow users to continue in the event of a false positive (i.e. when a certificate error is unlikely to result in a security compromise). In our study we examined these goals by observing whether participants discontinued visiting the bank website while continuing to the library website. These results from our laboratory experiment are displayed in Table 5. Participants who saw our single-page or multi-page warnings were more likely to heed the warnings than participants who saw the FF2 or IE7 warnings, but not the FF3 warning. In contrast, participants who saw our multi-page warning were more likely to visit the library website than participants who saw the FF3 warning. In the rest of this section we discuss demographics, present more detailed comparisons of the conditions and tasks, and present interesting qualitative results from our exit survey.

4.2.1 Participant Characteristics

We did not find any statistically significant demographic imbalances between participants in our randomly assigned conditions. The factors we tested were gender, nationality, age, technical sophistication, and a metric we call “cyberthreat exposure” designed to measure participants' prior experiences

	FF2		FF3		IE7		Single-Page	Multi-Page
Bank	18	(90%)	11	(55%)	18	(90%)	9 (45%)	12 (60%)
Library	19	(95%)	12	(60%)	20	(100%)	16 (80%)	19 (95%)

Table 5: Number (and percentage) of participants in each condition who ignored the warning and used the website to complete the library and bank tasks.

with information theft and fraud. Most demographic factors were determined by a single exit survey question (e.g. gender, nationality). Technical sophistication was measured by a composite score of five questions, the same as in the online survey. Similarly, cyberthreat exposure was measured by asking participants if they have ever had any account information stolen, found fraudulent transactions on bank statements, had a social security number stolen, or if they had ever been notified that personal information had been stolen or compromised.

Our participants were technically sophisticated, mostly male, and mostly foreign students. We had 68 male and only 32 female participants. All of our participants were between the ages of 18–30, and all but two were students. Sixty-nine participants were born in India, 17 in the United States, and the remaining were from Asia (10) and Europe (4). The average tech score was 1.90, which is significantly larger than the 0.66 average among the survey respondents.

We do not have a large enough sample size to determine whether age, profession, or nationality influenced participant behavior. In addition, our participants had so little cyberthreat exposure—83 participants answered affirmatively to 0 out of 4 questions—that we could not determine if exposure correlated with our results. On the other hand, while our sample was large enough to observe behavioral differences based on gender and technical sophistication if large differences existed, we observed no statistical differences in participant behavior based on those factors. Finally, we found no statistical difference in behavior based on task order in any of the conditions.

4.2.2 Effect of Warning Design on Behavior

Our study focused on evaluating whether SSL warnings effectively prevent users from transmitting sensitive information to suspicious websites, while allow-

ing them to continue in the event of a false positive.

We hypothesized that participants visiting the bank website who see our redesigned warnings would be significantly more likely to discontinue than participants who see the other warnings. We used a one-tailed Fisher’s exact test to analyze our results. We found that significantly more participants obeyed our single page warning than obeyed the FF2 and IE7 warnings ($p < 0.0029$ for both comparisons). Similarly, our multi-page warning performed better than the FF2 and IE7 warnings ($p < 0.0324$). However, FF3 was equivalently preventative, and it was also significantly better than the FF2 and IE7 warnings ($p < 0.0155$).

We also hypothesized that participants visiting the library website who see our redesigned warning will be significantly more likely to continue than participants who see the other warnings. In this case our hypothesis turned out to be mostly false. Participants who viewed our multi-page warning were significantly more likely to use the library website than participants who saw the FF3 warning ($p < 0.0098$). However, users of our multi-page warning visited the library website at an equal rate to users of the FF2 and IE7 warnings. Our single page warning was not significantly different than any of the other warnings. The FF3 warning caused significantly more participants to call the library than the FF2 warning ($p < 0.0098$) or the IE7 warning ($p < 0.0016$).

Two participants in the FF3 condition and one in our multi-page warning condition thought the library and bank servers were down or that we had blocked their websites. One wrote in the exit survey “the graphics made me feel the server was down” and another wrote “I just saw the title and assumed that it is just not working on this computer.” We suspect that users confuse the warnings with a 404 or server not found error, like the one shown in Figure 5. The

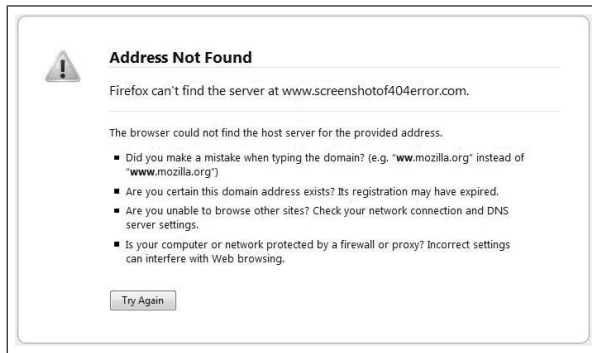


Figure 5: Screenshot of server not found error in FF3.

warnings have very similar layouts and coloring. The yellow Larry icon in the FF3 warning (Figure 3) and the first page of our multi-page (Figure 4(a)) warning is similar to the yellow triangle in Figure 5.

We took careful note of how participants in the multi-page warning condition answered the question “What type of website are you trying to visit?” presented to them on the first page of the warning. Fifteen participants answered exactly as expected – they selected “other” for the library and “bank or other financial institution” for the bank. The remaining five participants exhibited noteworthy behaviors: one participant did not answer the question for either task, while three participants performed the library task first and appropriately answered “other,” but also inaccurately answered “other” when visiting the bank website. This is stark evidence of the ill-effects of warning habituation – these participants learned how to ignore the warning in the library task and immediately reapplied their knowledge to the bank task. Finally, one participant first performed the bank task and correctly answered “bank or other financial institution.” However, when she saw the second page of the warning she clicked the back button and changed her answer to “other.”

4.2.3 Risk Perception in Context

We hypothesized that participants who viewed our multi-page warning would be more likely to obey the warnings when they were visiting the bank website than when they were visiting the library web-

site. Because this warning took context into account in determining severity, it appeared to be more severe on the bank website. All 14 participants in our study who heeded the library warning also heeded the warning at the bank. An additional 18 participants heeded the bank warning and proceeded past the library warning. Participants who viewed our multi-page warning ($p < 0.0098$) and our single-page warning ($p < 0.0242$) were significantly more likely to heed the warning at the bank than at the library.

We believe the behavior exhibited by users of our single page warning can be explained both by its success in raising awareness of risk and its clear communication of what users should do in response to the risk. When the 11 participants who heeded the single-page bank warning were asked in the exit survey “Why did you choose to heed or ignore the warning?” 9 out of 11 specifically mentioned the security of their information as the reason. In contrast only 2 participants in each of the FF2, FF3, and IE7 conditions mentioned risk in response to the same question. In addition, 10 of the 20 participants in our single-page warning condition when asked, “What action(s) did you think the warning at the bank wanted you to take?” responded that it wanted them *not* to proceed. Only 3 FF2, 2 FF3, and 4 IE7 participants answered the same way.

4.2.4 Impact of Reading and Understanding

In each of the first two sections of the exit survey we asked participants if they “read the text of the warning at the *bank/library* website.” At the bank website, significantly more people read our multi-page warning than the FF2 ($p < 0.0128$), FF3 ($p < 0.0018$), or IE7 ($p < 0.0052$) warnings (Table 6). There were no other significant differences in reported readership across conditions or tasks. We used a chi-square test to see if there was a difference in how reading affected behavior. Among the participants who did not read the warnings, FF2 and IE7 users were significantly more likely to log in to the bank website ($\chi^2_4 = 13.56$, $p < 0.009$), whereas FF3 users were significantly less likely to log in to the library website ($\chi^2_4 = 18.38$, $p < 0.001$).

The exit survey asked participants “what did

Condition	Read		Didn't Read		Understood		Didn't Understand	
	Logged In	Called	Logged In	Called	Logged In	Called	Logged In	Called
FF2	4	2	14	0	7	2	11	0
FF3	2	2	9	7	4	2	7	7
IE7	4	1	14	1	8	2	10	0
Single-Page	4	6	5	5	4	7	5	4
Multi-Page	8	6	4	2	7	6	5	2

Table 6: Behavior in the bank task by reading, understanding, and condition.

you believe the warning at the *bank/library* website meant?” Answers were entered into a free response text box and we categorized the responses according to whether or not they demonstrated understanding of the warning, as we had done in the survey (Table 6). In particular, participants who wrote that their connection may be compromised or that the identity of the destination website could not be verified were deemed to understand the warning. All other responses were coded as not understanding the meaning. There were no significant differences in the number of participants who understood the warnings based on condition in either task. However, participants in the FF3 condition who did not understand the warning were significantly more likely to call than users in the FF2 ($p < 0.0078$) and IE7 ($p < 0.0188$) conditions. Seven of the 14 participants who did not understand the FF3 warning called the bank. This is evidence that the FF3 users may have been prevented from visiting the websites because they did not know how to override warnings, and not because they understood the risks of proceeding.

One expects that participants who claimed to have read the warnings would be more likely to understand their meaning. When we combined the data from just our two warnings, single-page and multi-page, we found a statistically significant correlation ($p < 0.020$). However, we do not have enough data to determine whether there is a correlation for the three native warnings (FF2, FF3, and IE7).

4.2.5 Other Observations

One worry for browser designers trying to design effective warnings is that they will cause users to switch browsers, in favor of a browser that shows a less se-

Response	FF2	FF3	IE7	Single	Multi
Yes	8	7	10	4	1
No	8	11	5	16	16
Unknown	4	2	5	0	3

Table 7: Number of participants in each condition who claimed to have seen the warning before at the bank.

vere warning. In fact, during our study a few participants who viewed our warnings or the FF3 warnings asked or attempted to perform one of the tasks in a different browser. We directed them to continue using the browser they had been using. No participants in the FF2 and IE7 conditions tried to switch browsers. This indicates that complex warning designs may cause a small number of users to switch browsers. Therefore, for the sake of these users' security, it may be best if all browsers converged on a single warning design.

Among our strangest results were the answers to the questions: “Before this study, had you ever seen the warning you saw at the *bank/library* web site?” (Table 7). A total of 30 participants said they had seen the warning before at the bank website compared to only 16 at the library website. In addition, 5 participants in the bank task thought they had seen our warnings before. We do not think 30% of our participants have been scammed by man-in-the-middle attacks at their bank and we know for sure that the 5 participants had never seen our redesigned warnings before. This is dramatic evidence of memory problems, warning confusion, and general confusion with regard to certificate errors. At the same time, it is possible that the novelty of our new warnings

contributed to more participants reading them (and consequently better understanding the risks of ignoring them). None of the participants who viewed our new warnings could have seen them before, while our randomized condition assignments resulted in the two Firefox conditions being assigned 27 participants who were pre-existing Firefox users (68% of 40) and the IE condition being assigned 6 participants who were existing IE users (30% of 20). Thus, it is likely that these 33 participants had already been exposed to the warnings prior to our study, but among our sample population we observed no significant differences in behavior among them and the participants in the IE and FF conditions who were accustomed to using different browsers.

In the exit survey we asked participants to use a 7-point Likert scale to report the influence of several factors on their decision to heed or ignore the warnings. The factors we included were: the text of the warning, the colors of the warning, the choices that the warning presented, the destination URL, and the look and feel of the destination website. We expected significantly more participants to grade the color and text of the website highly for our warnings. However, there was no statistically significant difference in participants' responses based on condition.

5 Discussion

Our warnings somewhat improved user behavior, but all warning strategies, including ours, leave too many users vulnerable to man-in-the-middle attacks. The five warnings we evaluated embodied three different strategies: explain the potential danger facing users, make it difficult for users to ignore, and ask a question users can answer. The strategies have differences that we will discuss later in this section. However, regardless of how compelling or difficult to ignore, users think SSL warnings are of little consequence because they see them at legitimate websites. Many users have a completely backward understanding of the risk of man-in-the-middle attacks and assume that they are *less* likely to occur at trusted websites like those belonging to banks. If they do become fraud victims, they are unlikely to pinpoint it to their decision to

ignore a warning. Thus users' attitudes and beliefs about SSL warnings are likely to undermine their effectiveness [3]. Therefore, the best avenue we have for keeping users safe may be to avoid SSL warnings altogether and *really* make decisions for users—blocking them from unsafe situations and remaining silent in safe situations.

5.1 Limitations

We did not attempt to measure any long term affects of habituation to warnings. Many participants were likely to have previously seen the FF2 and IE7 warnings, while few users were likely to have seen FF3 warnings as that browser was released just before the study began. Our two warnings were new to all participants. We expect users were more likely to ignore the IE7 and FF2 warnings because of habituation, but this is not supported by our data.

Several artifacts of the study design may have caused participants to behave less securely than they normally would. Our study participants knew in advance that they would be using their bank credentials during the study and therefore the most security conscious potential participants may have decided not to perform the study. In addition, the study was performed at and sanctioned by Carnegie Mellon, and therefore participants may have trusted that the study would not put their credentials at risk.

In our study, users were much less likely to heed certificate warnings than in a previous study by Schechter et al. that also examined user responses to the IE7 certificate warning [17]. In our study 90% of participants ignored the IE7 warning while in the Schechter et al. study only 36% of participants who used their own accounts ignored the IE7 warning. We believe the differences may be due to the fact that in the previous study participants were told the study was about online banking, they performed four banking tasks prior to observing the warning, and they were given two other clues that the website might be insecure prior to the display of the warnings. The authors state, "responses to these clues may have been influenced by the presence of prior clues." Furthermore, the previous study was conducted while IE7 was still in beta and thus users were less likely to

have seen the certificate warning before. In addition, our study participants were more technically sophisticated than the previous study's participants.

5.2 Explain the Danger

The FF2, IE7, and our single page warnings take the standard tactic of explaining the potential danger to users. The FF2 warning, which is an unalarming popup box with obscure language, prevented very few users from visiting the bank or library. The IE7 warning, which has clearer language and a more frightening overall look, does not perform any better. On the other hand, our single page warning, with its black and red colors, was the most effective of the five warnings at preventing users from visiting the bank website. In addition, only four users called the library, indicating that our single-page warning would be only a minor nuisance for legitimate websites. That said, we suspect our single page warning would become less effective as users are habituated to it when visiting legitimate websites.

5.3 Make it Difficult

The FF3 warning, as discussed at length in Section 4.2.2, prevents user from visiting websites with invalid certificates by confusing users and making it difficult for them to ignore the warning. This improves user behavior in risky situations like the bank task, but it presents a significant nuisance in safer situations like the library task. Many legitimate websites that use self-signed certificates have posted online tutorials teaching users how to override the FF3 warning.¹² We suspect that users who learn to use the warning from these tutorials, by simple trial and error, help from a friend, etc., will ignore subsequent warnings and will be left both annoyed and unprotected.

¹²See for example: 1) http://hasylab.desy.de/infrastructure/experiment_control/links_and_tutorials/ff3_and_ssl/index_eng.html, 2) <http://www.engr.colostate.edu/webmail/>, and 3) http://knowledgehub.zeus.com/faqs/2008/02/05/configuring_zxtm_with_firefox_3

5.4 Ask a Question

Our multi-page warning, introduced in Section 4.1.2, asks the user a question in order to collect contextual information to allow the browser to better assess the risk of letting the user proceed to the website. This warning suffers from two usability problems: users may answer incorrectly because they are confused and users may knowingly answer incorrectly to get around the warning. In addition, it leaves users susceptible to active attacks such as the finer-grained origins attacks [9]. These problems, plus the fact that the single-page warning was more successful in preventing users from visiting the bank website, lead us to recommend against our multi-page warning as it is currently implemented.

The multi-page warning depends on users correctly answering our question, but only fifteen of the 20 participants answered correctly at the bank website. As discussed in Section 4.2.2, we believe that five participants either knowingly gave the wrong answer in order to reach the destination website without interruption, or they confused the warning with a server unavailable error. However, many users still made mistakes even when answering our question correctly. They behaved no more securely than users of our single-page warning.

Users who answered our question correctly and followed its advice would still be susceptible to finer-grained origins attacks. As brought to our attention by an anonymous reviewer, an attacker with control over the network or DNS may circumvent the multi-page warning by forcing the browser to connect to a website other than the one the user intended. For example, let's say Alice goes to a webmail site (www.mail.com), but an attacker controls the network and wants to steal the password to her online bank (www.bank.com).

When Alice visits mail.com, the attacker sends a response to the Alice that forwards the browser to <https://www.bank.com/action.js>. Then, the attacker intercepts the connection to the bank with a self-signed certificate, which triggers the warning shown in Figure 4(a). The warning asks her what type of website she is trying to reach and she answers "other" because she believes she is visiting her webmail. Since

Alice answered “other” she is immediately forwarded to `action.js`. If Alice has an open session with the bank, the attacker steals her `bank.com` secure cookies with the script.

Even if Alice does not have an open session with the bank, the browser’s cache will store the attack script. Let’s say in its normal operation the bank site loads its version of `action.js` after a user logs-in. (If the site loads a different script, then the attacker simply poisons that script instead.) If Alice logs-into `www.bank.com` in the next year, then the attacker’s version of `action.js` will load instead of the bank’s version. As in the attack in the previous paragraph, the script steals her secure cookies. There are many other variations on this attack, but they all rely on Alice answering “what type of website are you trying to visit” based on the site she believes she is visiting instead of the site the attacker sends to her.

Designing an interface to collect contextual information from users without making them susceptible to active attacks such as those outlined above poses a challenge. While we can ask users simple questions about their intentions that they are capable of answering, we must be sure that attackers cannot intervene to mislead users. We may be able to improve the multi-page warning we proposed by asking users another question in certain circumstances. In particular, if the URL of the connecting website is substantially different than the URL the user typed (or clicked on, in the case of a link), then we would show the URL of the connecting website and ask the user if they intended to visit that URL. Unfortunately this is not a complete solution for websites with mixed content, like those using a third-party shopping cart provider. In addition, the usability of such a solution remains untested.

It remains an open research challenge to determine how to leverage contextual information—including user-provided information—in order to assess risks. In particular, an approach is needed that is not vulnerable to confused users, users trying to get around the system, or active attackers. It remains to be seen whether it is feasible to design a robust approach that uses user-provided information. Alternative approaches may leverage contextual information provided by sources other than the user.

5.5 Avoid Warnings

The ideal solution to SSL warning problems is to block access when users are in true danger and allow users to proceed when they are not. This ideal is probably unattainable, but two systems recently presented by the research community, ForceHTTPS [10] and Perspectives [20] (and discussed in Section 2), are steps in the right direction. Both systems identify websites likely to be unsafe and use warnings to stop users from proceeding. It would be better to block these unsafe websites entirely. We expect both systems to have extremely low false positive rates, but further evaluation is required to know for sure. Another possible way of identifying unsafe websites is to maintain a list of websites that are verified by a root certificate authority and block websites on the list when the browser receives a self-signed certificate instead.

6 Acknowledgements

Thanks to Dhruv Mohindra, Amit Bhan, and Stuart Schechter for their help in the early stages of this project. This work was supported in part by Microsoft Research and by the National Science Foundation under Grants No. 0524189 and 0831428. The first author is supported by a National Defense Science and Engineering Graduate Fellowship.

References

- [1] J. C. Brustoloni and R. Villamarín-Salomón. Improving security decisions with polymorphic and audited dialogs. In *Proceedings of the 3rd symposium on Usable privacy and security*, pages 76–85, New York, NY, USA, 2007. ACM Press.
- [2] Certification Authority/Browser Forum. Extended validation SSL certificates, Accessed: July 27, 2007. <http://cabforum.org/>.
- [3] L. F. Cranor. A framework for reasoning about the human in the loop. In *Proceedings of the 1st Conference on Usability, Psychology, and Security*, pages 1–15, Berkeley, CA, USA, 2008. USENIX Association.
- [4] R. Dhamija, J. D. Tygar, and M. Hearst. Why phishing works. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 581–590, New York, NY, USA, 2006. ACM.

- [5] I. E-Soft. SSL server survey, February 1, 2007. http://www.securityspace.com/s_survey/sdata/200701/certca.html.
- [6] S. Egelman, L. F. Cranor, and J. Hong. You've been warned: an empirical study of the effectiveness of web browser phishing warnings. In *Proceeding of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1065–1074, New York, NY, USA, 2008. ACM.
- [7] B. Fogg, J. Marshall, O. Laraki, A. Osipovich, C. Varma, N. Fang, J. Paul, A. Rangeekar, J. Shon, P. Swani, and M. Treinen. What makes web sites credible? a report on a large quantitative study. In *Proceedings of the SIGCHI Conference on in Computing Systems*, Seattle, WA, March 31 - April 4, 2001. ACM.
- [8] B. Friedman, D. Hurley, D. C. Howe, E. Felten, and H. Nissenbaum. Users' conceptions of web security: a comparative study. In *Extended Abstracts on Human Factors in Computing Systems*, pages 746–747, New York, NY, USA, 2002. ACM.
- [9] C. Jackson and A. Barth. Beware of finer-grained origins. In *Proceedings of the Web 2.0 Security and Privacy Workshop*, 2008.
- [10] C. Jackson and A. Barth. ForceHTTPS: protecting high-security web sites from network attacks. In *Proceeding of the 17th International World Wide Web Conference*, pages 525–534, New York, NY, USA, 2008. ACM.
- [11] C. Jackson, D. R. Simon, D. S. Tan, and A. Barth. An evaluation of extended validation and picture-in-picture phishing attacks. In *Proceeding of the 1st International Workshop on Usable Security*, pages 281–293, Berlin / Heidelberg, Germany, February 2007. Springer.
- [12] S. Milgram. *Obedience to Authority: An Experimental View*. Harpercollins, 1974.
- [13] J. Nightingale. SSL information wants to be free, January 2009. <http://blog.johnath.com/2009/01/21/ssl-information-wants-to-be-free/>.
- [14] A. Patrick. Commentary on research on new security indicators. Self-published Online Essay, Accessed: January 15, 2009. <http://www.andrewpattick.ca/essays/commentary-on-research-on-new-security-indicators/>.
- [15] R. Rasmussen and G. Aaron. Global phishing survey: Domain name use and trends 1h2008. Anti-Phishing Working Group Advisory, November 2008. http://www.antiphishing.org/reports/APWG_GlobalPhishingSurvey1H2008.pdf.
- [16] B. Ross. Firefox and the worry free web. In L. F. Cranor and S. Garfinkel, editors, *Security and Usability: Designing Secure Systems that People Can Use*, pages 577–588. O'Reilly Media, Inc., Sebastopol, CA, USA, August 2005.
- [17] S. E. Schechter, R. Dhamija, A. Ozment, and I. Fischer. The emperor's new security indicators. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, pages 51–65, Washington, DC, USA, 2007. IEEE Computer Society.
- [18] J. Sobey, R. Biddle, P. C. van Oorschot, and A. S. Patrick. Exploring user reactions to new browser cues for extended validation certificates. In *Proceedings of the 13th European Symposium on Research in Computer Security*, pages 411–427, 2008.
- [19] D. W. Stewart and I. M. Martin. Intended and unintended consequences of warning messages: A review and synthesis of empirical research. *Journal of Public Policy & Marketing*, 13(1):1–1, 1994.
- [20] D. Wendlandt, D. G. Andersen, and A. Perrig. Perspectives: Improving SSH-style host authentication with multi-path probing. In *Proceedings of the 2008 USENIX Annual Technical Conference*, Berkeley, CA, USA, June 2008. USENIX Association.
- [21] T. Whalen and K. M. Inkpen. Gathering Evidence: Use of Visual Security Cues in Web Browsers. In *Proceedings of the 2005 Conference on Graphics Interface*, pages 137–144, Victoria, British Columbia, 2005.
- [22] M. Wogalter. Purpose and scope of warnings. In M. Wogalter, editor, *Handbook of Warnings*, pages 3–9. Lawrence Erlbaum Associates, Mahway, NJ, USA, 2006.
- [23] M. Wu, R. C. Miller, and S. L. Garfinkel. Do security toolbars actually prevent phishing attacks? In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 601–610, New York, NY, USA, 2006. ACM.
- [24] H. Xia and J. C. Brustoloni. Hardening web browsers against man-in-the-middle and eavesdropping attacks. In *Proceedings of the 14th International World Wide Web Conference*, pages 489–498, New York, NY, USA, 2005. ACM.

The Multi-Principal OS Construction of the Gazelle Web Browser

Helen J. Wang*, Chris Grier†, Alexander Moshchuk‡, Samuel T. King†, Piali Choudhury*, Herman Venter*

*Microsoft Research †University of Illinois at Urbana-Champaign ‡University of Washington

{helenw,pialic,hermanv}@microsoft.com, {grier,kingst}@uiuc.edu, anm@cs.washington.edu

Abstract

Original web browsers were applications designed to view static web content. As web sites evolved into dynamic web applications that compose content from multiple web sites, browsers have become multi-principal operating environments with resources shared among mutually distrusting web site *principals*. Nevertheless, no existing browsers, including new architectures like IE 8, Google Chrome, and OP, have a multi-principal operating system construction that gives a browser-based OS the *exclusive* control to manage the protection of all system resources among web site principals.

In this paper, we introduce Gazelle, a secure web browser constructed as a multi-principal OS. Gazelle's browser kernel is an operating system that *exclusively* manages resource protection and sharing across web site principals. This construction exposes intricate design issues that no previous work has identified, such as cross-protection-domain display and events protection. We elaborate on these issues and provide comprehensive solutions.

Our prototype implementation and evaluation experience indicates that it is realistic to turn an existing browser into a multi-principal OS that yields significantly stronger security and robustness with acceptable performance.

1 Introduction

Web browsers have evolved into a multi-principal operating environment where a principal is a web site [43]. Similar to a multi-principal OS, recent proposals [12, 13, 23, 43, 46] and browsers like IE 8 [34] and Firefox 3 [16] advocate and support programmer abstractions for protection (e.g., `<sandbox>` in addition to `<iframe>` [43]) and cross-principal communication (e.g., `PostMessage` [24, 43]). Nevertheless, no existing browsers, including new architectures like IE 8 [25], Google Chrome [37], and OP [21], have a multi-principal OS construction that gives a browser-based OS, typically called the browser kernel, the *exclusive* control to manage the protection and fair sharing of all system resources among browser principals.

In this paper, we present a multi-principal OS construction of a secure web browser, called Gazelle. Gazelle's browser kernel *exclusively* provides cross-principal protection and fair sharing of *all* system re-

sources. In this paper, we focus only on resource protection in Gazelle.

In Gazelle, the browser kernel runs in a separate protection domain (an OS process in our implementation), interacts with the underlying OS directly, and exposes a set of system calls for web site principals. We use the same web site principal as defined in the same-origin policy (SOP), which is labeled by a web site's origin, the triple of `<protocol, domain name, port>`. In this paper, we use "principal" and "origin" interchangeably. Unlike previous browsers, Gazelle puts web site principals into separate protection domains, completely segregating their access to all resources. Principals can communicate with one another only through the browser kernel using inter-process communication. Unlike all existing browsers except OP, our browser kernel offers the same protection to plugin content as to standard web content.

Such a multi-principal OS construction for a browser brings significant security and reliability benefits to the overall browser system: the compromise or failure of a principal affects that principal alone, leaving other principals and the browser kernel unaffected.

Although our architecture may seem to be a straightforward application of multi-principal OS construction to the browser setting, it exposes intricate problems that did not surface in previous work, including display protection and resource allocation in the face of cross-principal web service composition common on today's web. We will detail our solutions to the former and leave the latter as future work.

We have built an Internet-Explorer-based prototype that demonstrates Gazelle's multi-principal OS architecture and at the same time uses all the backward-compatible parsing, DOM management, and JavaScript interpretation that already exist in IE. Our prototype experience indicates that it is feasible to turn an existing browser into a multi-principal OS while leveraging its existing capabilities.

With our prototype, we successfully browsed 19 out of the top 20 Alexa-reported popular sites [5] that we tested. The performance of our prototype is acceptable, and a significant portion of the overhead comes from IE instrumentation, which can be eliminated in a production implementation.

We expect that the Gazelle architecture can be made fully backward compatible with today's web. Neverthe-

less, it is interesting to investigate the compatibility cost of eliminating the insecure policies in today's browsers. We give such a discussion based on a preliminary analysis in Section 9.

For the rest of the paper, we first give an in-depth comparison with related browser architectures in Section 2. We then describe Gazelle's security model 3. In Section 4, we present our architecture, its design rationale, and how we treat the subtle issue of legacy protection for cross-origin script source. In Section 5, we elaborate on the problem statement and design for cross-principal, cross-process display protection. We give a security analysis including a vulnerability study in Section 6. We describe our implementation in Section 7. We measure the performance of our prototype in Section 8. We discuss the tradeoffs of compatibility vs. security for a few browser policies in Section 9. Finally, we conclude and address future work in Section 10.

2 Related Work

In this section, we discuss related browser architectures and compare them with Gazelle.

2.1 Google Chrome and IE 8

In concurrent work, Reis *et al.* detailed the various process models supported by Google Chrome [37]: monolithic process, process-per-browsing-instance, process-per-site-instance, and process-per-site. A browsing instance contains all interconnected (or inter-referenced) windows including tabs, frames and subframes *regardless* of their origin. A site instance is a group of same-site pages within a browsing instance. A site is defined as a set of SOP origins that share a registry-controlled domain name: for example, *attackerAd.socialnet.com*, *alice.profiles.socialnet.com*, and *socialnet.com* share the same registry-controlled domain name *socialnet.com*, and are considered to be the same site or principal by Chrome. Chrome uses the process-per-site-instance model by default. Furthermore, Reis *et al.* [37] gave the caveats that Chrome's current implementation does *not* support strict site isolation in the process-per-site-instance and process-per-site models: embedded principals, such as a nested `iframe` sourced at a different origin from the parent page, are placed in the same process as the parent page.

The monolithic and process-per-browsing-instance models in Chrome do not provide memory or other resource protection across multiple principals in a monolithic process or browser instance. The process-per-site model does not provide failure containment across site instances [37]. Chrome's process-per-site-instance

model is the closest to Gazelle's two processes-per-principal-instance model, but with several crucial differences: (1) Chrome's principal is site (see above) while Gazelle's principal is the same as the SOP principal. (2) A web site principal and its embedded principals co-exist in the same process in Chrome, whereas Gazelle places them into separate protection domains. Pursuing this design led us to new research challenges including cross-principal display protection (Section 5). (3) Plugin content from different principals or sites share a plugin process in Chrome, but are placed into separate protection domains in Gazelle. (4) Chrome relies on its rendering processes to enforce the same-origin policy among the principals that co-exist in the same process. These differences indicate that in Chrome, cross-principal (or -site) protection takes place in its rendering processes and its plugin process, in addition to its browser kernel. In contrast, Gazelle's browser kernel functions as an OS, managing cross-principal protection on all resources, including display.

IE 8 [25] uses OS processes to isolate tabs from one another. This granularity is insufficient since a user may browse multiple mutually distrusting sites in a single tab, and a web page may contain an `iframe` with content from an untrusted site (e.g., ads).

Fundamentally, Chrome and IE 8 have different goals from that of Gazelle. Their use of multiple processes is for failure containment across the user's browsing sessions rather than for security. Their security goal is to protect the host machine from the browser and the web; this is achieved by process sandboxing [9]. Chrome and IE 8 achieved a good milestone in the evolution of the browser architecture design. Looking forward, as the world creates and migrates more data and functionality into the web and establishes the browser as a dominant application platform, it is critical for browser designers to think of browsers as operating systems and protect web site principals from one another in addition to the host machine. This is Gazelle's goal.

2.2 Experimental browsers

The OP web browser [21] uses processes to isolate browser components (i.e., HTML engine, JavaScript interpreter, rendering engine) as well as pages of the same origin. In OP, intimate interactions between browser components, such as JavaScript interpreter and HTML engine, must use IPC and go through its browser kernel. The additional IPC cost does not add much benefits: isolating browser components within an instance of a web page provides no additional security protection. Furthermore, besides plugins, basic browser components are fate-shared in web page rendering: the failure of any one browser component results in most web pages not

functioning properly. Therefore, process isolation across these components does not provide any failure containment benefits either. Lastly, OP's browser kernel does not provide all the cross-principal protection needed as an OS because it delegates display protection to its processes.

Tahoma [11] uses virtual machines to completely isolate (its own definition of) web applications, disallowing any communications between the VMs. A web application is specified in a manifest file provided to the virtual machine manager and typically contains a suite of web sites of possibly different domains. Consequently, Tahoma doesn't provide protection to existing browser principals. In contrast, Gazelle's browser kernel protects browser principals first hand.

The Building a Secure Web Browser project [27, 28] uses SubOS processes to isolate content downloading, display, and browser instances. SubOS processes are similar to Unix processes except that instead of a user ID, each process has a SubOS ID with OS support for isolation between objects with different SubOS IDs. SubOS instantiates a browser instance with a different SubOS process ID for each URL. This means that the principal in SubOS is labelled with the URL of a page (protocol, host name plus path) rather than the SOP origin as in Gazelle. Nevertheless, SubOS does not handle embedded principals, unlike Gazelle. Therefore, they also do not encounter the cross-principal display-sharing issue which we tackle in depth. SubOS's principal model would also require all cross-page interactions that are common within a SOP origin to go through IPC, incurring significant performance cost for many web sites.

3 Security model

3.1 Background: security model in existing browsers

Today's browsers have inconsistent access and protection model for various resources. These inconsistencies present significant hurdles for web programmers to build robust web services. In this section, we give a brief background on the relevant security policies in existing browsers. Michal Zalewski gives an excellent and perhaps the most complete description of existing browsers' security model to date [48].

Script. The same-origin policy (SOP) [39] is the central security policy on today's browsers. SOP governs how scripts access the HTML document tree and remote store. SOP defines the *origin* as the triple of `<protocol, domain-name, port>`. SOP mandates that two documents from different origins cannot access each other's HTML documents using the Document Object Model (DOM), which is the platform- and language-

neutral interface that allows scripts to dynamically access and update the content, structure and style of a document [14]. A script can access its document origin's remote data store using the XMLHttpRequest object, which issues an asynchronous HTTP request to the remote server [45]. (XMLHttpRequest is the cornerstone of AJAX programming.) SOP allows a script to issue an XMLHttpRequest only to its enclosing page's origin. A script executes as the principal of its enclosing page though its source code is not readable in a cross-origin fashion.

For example, an `<iframe>` with source `http://a.com` cannot access any HTML DOM elements from another `<iframe>` with source `http://b.com` and vice versa. `http://a.com`'s scripts (regardless of where the scripts are hosted) can issue XMLHttpRequests to only `a.com`. Furthermore, `http://a.com` and `https://a.com` are different origins because of the protocol difference.

Cookies. For cookie access, by default, the principal is the host name and path, but without the protocol [19, 32]. For example, if the page `a.com/dir/1.html` creates a cookie, then that cookie is accessible to `a.com/dir/2.html` and other pages from that directory and its subdirectories, but is not accessible to `a.com/`. Furthermore, `https://a.com/` and `http://a.com/` share the cookie store unless a cookie is marked with a "secure" flag. Non-HTTPS sites may still set secure cookies in some implementations, just not read them back [48]. A web programmer can make cookie access less restrictive by setting a cookie's domain attribute to a postfix domain or the path name to be a prefix path. The browser ensures that a site can only set its own cookie and that a cookie is attached only to HTTP requests to that site.

The path-based security policy for cookies does not play well with SOP for scripts: scripts can gain access to all cookies belonging to a domain despite path restrictions.

Plugins. Current major browsers do not enforce any security on plugins and grant plugins access to the local operating system directly. The plugin content is subject to the security policies implemented in the plugin software rather than the browser.

3.2 Gazelle's security model

Gazelle's architecture is centered around protecting principals from one another by separating their respective resources into OS-enforced protection domains. Any sharing between two different principals must be explicit using cross-principal communication (or IPC) mediated by the browser kernel.

We use the same principal as the SOP, namely, the triple of `<protocol, domain-name, port>`. While it is tempting to have a more fine-grained principal,

we need to be concerned with co-existing with current browsers [29, 43]: the protection boundary of a more fine-grained principal, such as a path-based principal, would break down in existing browsers. It is unlikely that web programmers would write very different versions of the same service to accommodate different browsers; instead, they would forego the more fine-grained principal and have a single code base.

The resources that need to be protected across principals [43] are memory such as the DOM objects and script objects, persistent state such as cookies, display, and network communications.

We extend the same principal model to all content types except scripts and style sheets (Section 4): the elements created by `<object>`, `<embed>`, ``, and certain types of `<input>`¹ are treated the same as an `<iframe>`: the origin of the included content labels the principal of the content. This means that we *enforce* SOP on plugin content². This is consistent with the existing movement in popular plugins like Adobe Flash Player [20]. Starting with Flash 7, Adobe Flash Player uses the exact domain match (as in SOP) rather than the earlier “superdomain” match (where *www.adobe.com* and *store.adobe.com* have the same origin) [2]; and starting with Flash 9, the default ActionScript behavior only allows access to same-origin HTML content unlike the earlier default that allows full cross-origin interactions [1].

Gazelle’s architecture naturally yields a security policy that partitions all system resources across the SOP principal boundaries. Such a policy offers consistency across various resources. This is unlike current browsers where the security policies vary for different resources. For example, cookies use a different principal than that of scripts (see the above section); descendant navigation policy [7, 8] also implicitly crosses the SOP principal boundary (more in Section 5.1).

It is feasible for Gazelle to enable the same security policies as the existing browsers and achieve backward compatibility through cross-principal communications. Nevertheless, it is interesting to investigate the tradeoffs between supporting backward compatibility and eliminating insecure policies in today’s browsers. We gave a preliminary discussion on this in Section 9.

4 Architecture

4.1 Basic Architecture

Figure 1 shows our basic architecture. A principal is the *unit of protection*. Principals need to be completely isolated in resource access and usage. Any sharing must

be made explicit. Just as in desktop applications, where instances of an application are run in separate processes for failure containment and independent resource allocation, a principal instance is the *unit of failure containment* and the *unit of resource allocation*. For example, navigating to the same URL in different tabs corresponds to two instances of the same principal; when *a.com* embeds two *b.com* iframes, the *b.com* iframes correspond to two instances of *b.com*. However, the frames that share the same origin as the host page are in the same principal instance as the host page by default, though we allow the host page to designate an embedded same-origin frame or object as a separate principal instance for independent resource allocation and failure containment. Principal instances are isolated for all runtime resources, but principal instances of the same principal share persistent state such as cookies and other local storage. Protection unit, resource allocation unit, and failure containment unit can each use a different mechanism depending on the system implementation. Because the implementation of our principal instances contains native code, we use OS processes for all three purposes.

Our principal instance is similar to Google Chrome’s site instance [37], but with two crucial differences: 1) Google Chrome considers the sites that share the same registrar-controlled domain name to be from the same site, so *ad.datacenter.com*, *user.datacenter.com*, and *datacenter.com* are considered to be the same site and belong to the same principal. In contrast, we consider them as separate principals. 2) When a site, say *a.com*, embeds another principal’s content, say an `<iframe>` with source *b.com*, Google Chrome puts them into the same site instance. In contrast, we put them into separate principal instances.

The browser kernel runs in a separate protection domain and interposes between browser principals and the traditional OS. The browser kernel mediates the principals’ access to system resources and enforces security policies of the browser. Essentially, the browser kernel functions as an operating system to browser principals and manages the protection and sharing of system resources for them. The browser kernel also manages the browser chrome, such as the address bar and menus. The browser kernel receives all events generated by the underlying operating system including user events like mouse clicks or keyboard entries; these events are then dispatched to the appropriate principal instance. When the user navigates a window by clicking on a hyperlink that points to an URL at a different origin, the browser kernel creates the protection domain for the URL’s principal instance (if one doesn’t exist already) to render the target page, destroys the protection domain of the hyperlink’s host page, and re-allocates and re-initializes the window to the URL’s principal instance. The browser

¹`<input>` can be used to include an image using a “src” attribute.

²OP [21] calls this plugin policy the *provider domain policy*.

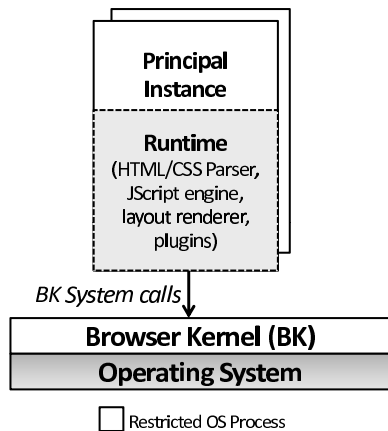


Figure 1: The Gazelle architecture

kernel is agnostic of DOM and content semantics and has a relatively simple logic.

The runtime of a principal instance performs content processing and is essentially an instance of today’s browser components including HTML and style sheet parser, JavaScript engine, layout renderer, and browser plugins. The only way for a principal instance to interact with system resources, such as networking, persistent state, and display, is to use browser kernel’s system calls. Principals can communicate with one another using message passing through the browser kernel, in the same fashion as inter-process communications (IPC).

It is necessary that the protection domain of a principal instance is a restricted or sandboxed OS process. The use of process guarantees the isolation of principals even in the face of attacks that exploit memory vulnerabilities. The process must be further restricted so that any interaction with system resources is limited to the browser kernel system calls. Native Client [47] and Xax [15] have established the feasibility of such process sandboxing.

This architecture can be efficient. By putting all browser components including plugins into one process, they can interact with one another through DOM intimately and efficiently as they do in existing browsers. This is unlike the OP browser’s approach [21] in which all browser components are separated into processes; chatty DOM interactions must be layered over IPCs through the OP browser kernel, incurring unnecessary overhead without added security.

Unlike all existing browsers except OP, this architecture can enforce browser security policies on plugins, namely, plugin content from different origins are segregated into different processes. Any plugin installed is unable to interact with the operating system and is only provided access to system resources subject to the browser kernel allowing that access. In this architecture, the payload that exploits plugin vulnerabilities will only com-

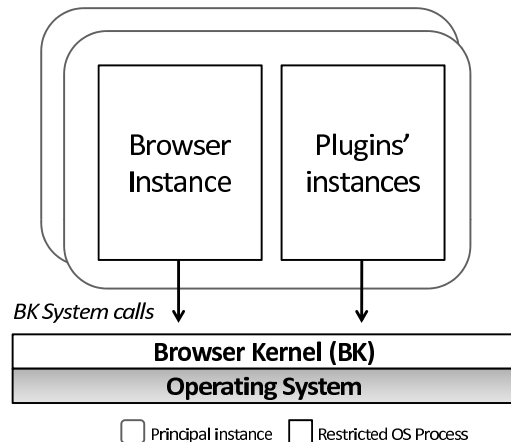


Figure 2: Supporting legacy protection

promise the principal with the same origin as the malicious plugin content, but not any other principals nor browser kernel.

The browser kernel supports the following system calls related to content fetching in this architecture (a more complete system call table is shown in Table 3):

- *getSameOriginContent (URL)*: Fetch the content at *URL* that has the same origin as the issuing principal regardless of the content type.
- *getCrossOriginContent (URL)*: Fetch the script or style sheet content from *URL*; *URL* may be from different origin than the issuing principal. The content type is determined by the *content-type* header of the HTTP response.
- *delegate (URL, windowSpec)*: Delegate a display area to a different principal of *URL* and fetch the content for that principal.

The semantics of these system calls is that the browser kernel can return cross-origin script or style content to a principal based on the *content-type* header of the HTTP response, but returns other content if and only if the content has the same origin as the issuing principal, abiding the same-origin policy. All the security decisions are made and enforced by the browser kernel alone.

4.2 Supporting Legacy Protection

The system call semantics in the basic architecture has one subtle issue: cross-origin script or style sheet sources are readable by the issuing principal, which does not conform with the existing SOP. The SOP dictates that a script can be executed in a cross-origin fashion, but the access to its source code is restricted to same origin only.

A key question to answer is that whether a script should be processed in the protection domain of its

provider (indicated in “src”), in the same way as frames, or in the protection domain of the host page that embeds the script. To answer this question, we must examine the primary intent of the script element abstraction. Script is primarily a *library* abstraction (which is a necessary and useful abstraction) for web programmers to include in their sites and runs with the privilege of the includer sites [43]. This is in contrast with the frame abstractions: Programmers put content into cross-origin frames so that the content runs as the principal of its own provider and be protected from other principals. Therefore, a script should be handled by the protection domain of its includer.

In fact, it is a flaw of the existing SOP to offer protection for cross-origin script source. Evidence has shown that it is extremely dangerous to hide sensitive data inside a script [22]. Numerous browser vulnerabilities exist for failing to provide the protection.

Unfortunately, web sites that rely on cross-origin script source protection, exist today. For example, GMail’s contact list is stored in a script file, at the time of writing. Furthermore, it is increasingly common for web programmers to adopt JavaScript Object Notation (JSON) [31] as the preferred data-interchange format. Web sites often demand such data to be same-origin access only. To prevent such data from being accidentally accessed through `<script>` (by a different origin), web programmers sometimes put “while (1);” prior to the data definition or put comments around the data so that accidental script inclusion would result in infinite loop execution or a no-op.

In light of the existing use, new browser architecture design must also offer the cross-origin script source protection. One way to do this is to strip all authentication-containing information, such as cookies and HTTP authentication headers, from the HTTP requests that retrieve cross-origin scripts so that the web servers will not supply authenticated data. The key problem with this approach is that it is not always clear what in an HTTP request may contain authentication information. For example, some cookies are used for authentication purposes and some are not. Stripping all cookies may impair functionality when the purpose of some cookies are not for authentication purposes. In another example, a network may use IP addresses for authentication, which are impossible to strip out.

We address the cross-origin script source protection problem by modifying our architecture slightly, as shown in Figure 2. The modification is based on the following observation. Third-party plugin software vulnerabilities have surged recently [36]. Symantec reports that in 2007 alone there are 467 plugin vulnerabilities [42], which is about one magnitude higher than that of browser software. Clearly, plugin software should be trusted much

less than browser software. Therefore, for protecting cross-origin script or style sheet source, we place more trust in the browser code and let the browser code retrieve and protect cross-origin script or style sheet sources: for each principal, we run browser code and plugin code in two separate processes. The plugin instance process *cannot* issue the `getCrossOriginContent()` and it can only interact with cross-origin scripts and style sheets through the browser instance process.

In this architecture, the quality of protecting cross-origin script and style-sheet source relies on the browser code quality. While this protection is not perfect with native browser code implementation, the architecture offers the same protection as OP, and stronger protection than the rest of existing browsers. The separation of browser code and plugin code into separate processes also improves reliability by containing plugin failures.

In recent work, Native Client [47] and Xax [15] have presented a plugin model that uses sandboxed processes to contain each browser principal’s plugin content. Their plugin model works perfectly in our browser architecture. We do not provide further discussions on plugins in our paper.

5 Cross-Principal, Cross-Process Display and Events Protection

Cross-principal service composition is a salient nature of the web and is commonly used in web applications. When building a browser as a multi-principal OS, this composition raises new challenges in display sharing and event dispatching: when a web site embeds a cross-origin frame (or objects, images), the involved principal instances share the display at the same time. Therefore, it is important that the browser kernel 1) discerns display and events ownership, 2) enforces that a principal instance can only draw in its own display areas, 3) dispatches UI events to only the principal instance with which the user is interacting. An additional challenge is that the browser kernel must accomplish these without access to any DOM semantics.

From a high level, in Gazelle principal instances are responsible for rendering content into bitmap objects, and our browser kernel manages these bitmap objects and chooses when and where to display them. Our architecture provides a clean separation between the act of rendering web content and the policies of how to display this content. This is a stark contrast to today’s browsers that intermingle these two functions, which has led to numerous security vulnerabilities [18, 44].

Our display management fundamentally differs from that of the traditional multi-user OSes, such as Unix and Windows. Traditional OSes offer no cross-principal dis-

play protection. In X, all the users who are authorized (through `.Xauthority`) to access the display can access one another's display and events. Experimental OSes like EROS [41] have dealt with cross-principal display protection. However, the browser context presents new challenges that are absent in EROS, such as dual ownership of display and cross-principal transparent overlays.

5.1 Display Ownership and Access Control

We define *window* to be a unit of display allocation and delegation. Each window is allocated by a *landlord* principal instance or the browser kernel; and each window is delegated to (or rented to) a *tenant* principal instance. For example, when the web site *a.com* embeds a frame sourced at *b.com*, *a.com* allocates a window from its own display area and delegates the window to *b.com*; *a.com* is the landlord of the newly-created window, while *b.com* is the tenant of that window. The same kind of delegation happens when cross-origin object and image elements are embedded. The browser kernel allocates top-level windows (or tabs). When the user launches a site through address-bar entry, the browser kernel delegates the top-level window to the site, making the site a tenant. We decided against using “parent” and “child” terminologies because they only convey the window hierarchy, but not the principal instances involved. In contrast, “landlord” and “tenant” convey both semantics.

Window creation and delegation result in a `delegate(URL, position, dimensions)` system call. For each window, the browser kernel maintains the following state: its landlord, tenant, position, dimensions, pixels in the window, and the URL location of the window content. The browser kernel manages a three-dimensional display space where the position of a window also contains a stacking order value (toward the browsing user). A landlord provides the stacking order of all its delegated windows to the browser kernel. The stacking order is calculated based on the DOM hierarchy and the CSS z-index values of the windows.

Because a window is created by a landlord and occupied by a tenant, the browser kernel must allow reasonable window interactions from both principal instances without losing protection. When a landlord and its tenant are from different principals, the browser kernel provides access control as follows:

- *Position and dimensions*: When a landlord embeds a tenant's content, the landlord should be able to retain control on what gets displayed on the landlord's display and a tenant should not be able to reposition or resize the window to interfere with the landlord's display. Therefore, the browser kernel enforces that only the landlord of a window can change the position and the dimensions of a window.

	Landlord	Tenant
position (x,y,z)	RW	
dimensions (height, width)	RW	R
pixels		RW
URL location	W	RW

Table 1: Access control policy for a window's landlord and tenant

- *Drawing isolation*: Pixels inside the window reflect the tenant's private content and should not be accessible to the landlord. Therefore, the browser kernel enforces that only the tenant can draw within the window. (Nevertheless, a landlord can create overlapping windows delegated to different principal instances.)
- *Navigation*: Setting the URL location of a window navigates the window to a new site. Navigation is a fundamental element of any web application. Therefore, both the landlord and the tenant are allowed to set the URL location of the window. However, the landlord should not obtain the tenant's navigation history that is private to the tenant. Therefore, the browser kernel prevents the landlord from reading the URL location. The tenant can read the URL location as long as it remains being the tenant. (When the window is navigated to a different principal, the old tenant will no longer be associated with the window and will not be able to access the window's state.)

Table 1 summarizes the access control policies in the browser kernel. In existing browsers, these manipulation policies also vaguely exist. However, their logic is intermingled with the DOM logic and is implemented at the object property and method level of a number of DOM objects which all reside in the same protection domain despite their origins. This had led to numerous vulnerabilities [18, 44]. In Gazelle, by separating these security policies from the DOM semantics and implementation, and concentrating them inside the browser kernel we achieve more clarity in our policies and much stronger robustness of our system construction.

The browser kernel ensures that principal instances other than the landlord and the tenant cannot manipulate any of the window states. This includes manipulating the URL location for navigation. Here, we depart from the existing descendant navigation policy in most of today's browsers [7, 8]. Descendant navigation policy allows a landlord to navigate a window created by its tenant even if the landlord and the tenant are different principals. This is flawed in that a tenant-created window is a resource that belongs to the tenant and should not be controllable by a different principal.

Existing literature [7, 8] supports the descendant navigation policy with the following argument: since existing browsers allow the landlord to draw over the tenant, a landlord can simulate the descendant navigation by overdrawing. Though overdrawing can *visually* simulate navigation, navigation is much more powerful than overdrawing because a landlord with such descendant navigation capability can interfere with the tenant's operations. For example, a tenant may have a script interacting with one of its windows and then effecting changes to the tenant's backend; navigating the tenant's window requires just one line of JavaScript and could effect undesirable changes in the tenant's backend. With overdrawing, a landlord can imitate a tenant's content, but the landlord cannot send messages to the tenant's backend in the name of the tenant.

5.2 Cross-Principal Events Protection

The browser kernel captures all events in the system and must accurately dispatch them to the right principal instance to achieve cross-principal event protection. Networking and persistent-state events are easy to dispatch. However, user interface events pose interesting challenges to the browser kernel in discerning event ownership, especially when dealing with overlapping, potentially transparent cross-origin windows: major browsers allow web pages to mix content from different origins along the z-axis where content can be occluded, either partially or completely, by cross-origin content. In addition, current standards allow web pages to make a frame or portions of their windows transparent, further blurring the lines between principals. Although these flexible mechanisms have a slew of legitimate uses, they can be used to fool users into thinking they are interacting with content from one origin, but are in fact interacting with content from a different origin. Zalewski [48] gave a taxonomy on "UI redressing" or clickjacking attacks which illustrated some of the difficulties with current standards and how attackers can abuse these mechanisms.

To achieve cross-principal events protection, the browser kernel needs to determine the *event owner*, the principal instance to which the event is dispatched. There are two types of events for the currently active tab: stateless and stateful. The owner of a stateless event like a mouse event is the tenant of the window (or display area) on which the event takes place. The owner of a stateful event such as a key-press event is the tenant of the current in-focus window. The browser kernel interprets mouse clicks as focus-setting events and keeps track of the current in-focus window and its principal instance.

The key problem to solve then is to determine the window on which a stateless or focus-setting event takes place. We consider a determination to have high *fidelity*

if the determined event owner corresponds to the user intent. Different window layout policies directly affect the fidelity of this determination. We elaborate on our explorations of three layout policies and their implications on fidelity.

Existing browsers' policy. The layout policy in existing browsers is to draw windows according to the DOM hierarchy and the z-index values of the windows. Existing browsers then associate a stateless or focus-setting event to the window that has the highest stacking order. Today, most browsers permit page authors to set transparency on cross-origin windows [48]. This ability can result in poor fidelity in determining the event owner in the face of cross-principal transparent overlays. When there are transparent, cross-origin windows overlapping with one another, it is impossible for the browser kernel to interpret the user's intent: the user is guided by what she sees on the screen; when two windows present a mixed view, some user interfaces visible to the user belong to one window, and yet some belong to another. The ability to overlay transparent cross-origin content can be extremely dangerous: a malicious site can make an iframe sourced at a legitimate site transparent and overlaid on top of the malicious site [48], fooling the users to interact with the legitimate site unintentionally.

2-D display delegation policy. This is a new layout policy that we have explored. In this policy, the display is managed as two-dimensional space for the purpose of delegation. Once a landlord delegates a rectangular area to a tenant, the landlord cannot overdraw the area. Thus, no cross-principal content can be overlaid. Such a layout constraint will enable perfect fidelity in determining an event ownership that corresponds to the user intent. It also yields better security as it can prevent all UI redressing attacks except clickjacking [48]. Even clickjacking would be extremely difficult to launch with this policy on our system since our cross-principal memory protection makes reading and writing the scrolling state of a window an exclusive right of the tenant of the window.

However, this policy can have a significant impact on backward compatibility. For example, a menu from a host page cannot be drawn over a nested cross-origin frame or object; many sites would have significant constraints with their own DOM-based pop-up windows created with `div`s and such (rather than using `window.open` or `alert`), which could overlay on cross-origin frames or objects with existing browsers' policy; and a cross-origin image cannot be used as a site's background.

Opaque overlay policy. This policy retains existing browsers' display management and layout policies as much as possible for backward compatibility (and additionally provides cross-principal events protection), but lets the browser kernel enforce the following layout invariant or constraint: for any two dynamic content-

containing windows (e.g., frames, objects) $win1$ and $win2$, $win1$ can overlay on $win2$ iff $(Tenant_{win1} == Tenant_{win2}) \vee (Tenant_{win1} \neq Tenant_{win2} \wedge win1 \text{ is opaque})$. This policy effectively constrains a pixel to be associated with just one principal, making event owner determination trivial. This is in contrast with the existing browsers' policy where a pixel may be associated with more than one principals when there are transparent cross-principal overlays. This policy allows same-origin windows to transparently overlay with one another. It also allows a page to use a cross-origin image (which is static content) as its background. Note that no principal instance other than the tenant of the window can set the background of a window due to our memory protection across principal instances. So, it is impossible for a principal to fool the user by setting another principal's background. The browser kernel associates a stateless event or a focus-setting event with the dynamic content-containing window that has the highest stacking order.

This policy eliminates the attack vector of overlaying a transparent victim page over an attacker page. However, by allowing overlapping opaque cross-principal frames or objects, it allows not only legitimate uses, such as those denied by the 2D display delegation policy, but it also allows an attacker page to cover up and expose selective areas of a nested cross-origin victim frame or object. The latter scenario can result in infidelity. We leave as future work the mitigation of such infidelity by determining how much of a principal's content is exposed in an undisturbed fashion to the user when the user clicks on the page.

We implemented the opaque overlay policy in our prototype.

6 Security Analysis

In Gazelle, the trusted computing base encompasses the browser kernel and the underlying OS. If the browser kernel is compromised, the entire browser is compromised. If the underlying OS is compromised, the entire host system is compromised. If the DNS is compromised, all the non-HTTPS principals can be compromised. When the browser kernel, DNS, and the OS are intact, our architecture guarantees that the compromise of a principal instance does not give it any capabilities in addition to those already granted to it through browser kernel system call interface (Section 4).

Next, we analyze Gazelle's security over classes of browser vulnerabilities. We also make a comparison with popular browsers with a study on their past, known vulnerabilities.

- Cross-origin vulnerabilities:

By separating principals into different protection domains and making any sharing explicit, we can much more easily eliminate cross-origin vulnerabilities. The only logic for which we need to ensure correctness is the origin determination in the browser kernel.

This is unlike existing browsers, where origin validations and SOP enforcement are spread through the browser code base [10], and content from different principals coexists in shared memory. All of the cross-origin vulnerabilities illustrated in Chen et al. [10] simply do not exist in our system; *no* special logic is required to prevent them because all of those vulnerabilities exploit implicit sharing.

Cross-origin script source can still be leaked in our architecture if a site can compromise its browser instance. Nevertheless, only that site's browser instance is compromised, while other principals are intact, unlike all existing browsers except OP.

- Display vulnerabilities:

The display is also a resource that Gazelle's browser kernel protects across principals, unlike existing browsers (Section 5). Cross-principal display and events protection and access control are enforced in the browser kernel. This prevents a potentially compromised principal from hijacking the display and events that belong to another principal. Display hijacking vulnerabilities have manifested themselves in existing browsers [17, 26] that allow an attacker site to control another site's window content.

- Plugin vulnerabilities:

Third-party plugins have emerged to be a significant source of vulnerabilities [36]. Unlike existing browsers, Gazelle's design requires plugins to interact with system resources only by means of browser kernel system calls so that they are subject to our browser's security policy. Plugins are contained inside sandboxed processes so that basic browser code doesn't share fate with plugin code (Section 4). A compromised plugin affects the principal instance's plugin process only, and not other principal instances nor the rest of the system. In contrast, in existing browsers except OP, a compromised plugin undermines the entire browser and often the host system as well.

A DNS rebinding attack results in the browser labeling resources from different network hosts with a common origin. This allows an attacker to operate within SOP and access unauthorized resources [30]. Although Gazelle does not fundamentally address this vulnerability, the fact that plugins must interact with the network through browser kernel system

	IE 7	Firefox 2
Origin validation error	6	11
Memory error	38	25
GUI logic flaw	3	13
Others	-	28
Total	47	77

Table 2: Vulnerability Study for IE 7 and Firefox 2

calls defeats the multipin form of such attacks.

We analyzed the known vulnerabilities of two major browsers, Firefox 2 [3] and IE 7 [35], since their release to November 2008, as shown in Table 2. For both browsers, memory errors are a significant source of errors. Memory-related vulnerabilities are often exploited by maliciously crafted web pages to compromise the entire browser and often the host machines. In Gazelle, although the browser kernel is implemented with managed C# code, it uses native .NET libraries, such as network and display libraries; memory errors in those libraries could still cause memory-based attacks against the browser kernel. Memory attacks in principal instances are well-contained in their respective sandboxed processes.

Cross-origin vulnerabilities, or origin validation errors, constitute another significant share of vulnerabilities. They result from the implicit sharing across principals in existing browsers and can be much more easily eliminated in Gazelle because cross-principal protection is exclusively handled by the browser kernel and because of Gazelle’s use of sandboxed processes.

In IE 7, there are 3 GUI logic flaws which can be exploited to spoof the contents of the address bar. For Gazelle, the address bar UI is owned and controlled by our browser kernel. We anticipate that it will be much easier to apply code contracts [6] in the browser kernel than in a monolithic browser to eliminate many of such vulnerabilities.

In addition, Firefox had other errors which didn’t map into these three categories, such as JavaScript privilege escalation, URL handling errors, and parsing problems. Since Gazelle enforces security properties in the browser kernel, any errors that manifest as the result of JavaScript handling and parsing are limited in the scope of exploit to the principal instance owning the page. URL handling errors could occur in our browser kernel as well.

7 Implementation

We have built a Gazelle prototype mostly as described in Section 4. We have not yet ported an existing plugin onto our system. Our prototype runs on Windows Vista with

.NET framework 3.5 [4]. We next discuss the implementation of two major components shown in Figure 2: the browser kernel and the browser instance.

Browser Kernel. The browser kernel consists of approximately 5k lines of C# code. It communicates with principal instances using system calls and upcalls, which are implemented as asynchronous XML-based messages sent over named pipes. An overview of browser kernel system calls and upcalls is presented in Table 3. System calls are performed by the browser instance or plugins and sometimes include replies. Upcalls are messages from the browser kernel to the browser instance.

Display management is implemented as described in Section 5 using .NET’s Graphics and Bitmap libraries. Each browser instance provides the browser kernel with a bitmap for each window of its rendered content using a display system call; each change in rendered content results in a subsequent display call. For each top-level browsing window (or tab), browser kernel maintains a stacking order and uses it to compose various bitmaps belonging to a tab into a single master bitmap, which is then attached to the tab’s PictureBox form. This straightforward display implementation has numerous optimization opportunities, many of which have been thoroughly studied [33, 38, 40], and which are not the focus of our work.

Browser instance. Instead of undertaking a significant effort of writing our own HTML parser, renderer, and JavaScript engine, we borrow these components from Internet Explorer 7 in a way that does not compromise security. Relying on IE’s Trident renderer has a big benefit of inheriting IE’s page rendering compatibility and performance. In addition, such an implementation shows that it is realistic to adapt an existing browser to use Gazelle’s secure architecture.

In our implementation, each browser instance embeds a Trident *WebBrowser* control wrapped with an *interposition layer* which enforces Gazelle’s security properties. The interposition layer uses Trident’s COM interfaces, such as *IWebBrowser2* or *IWebBrowserEvents2*, to hook sensitive operations, such as navigation or frame creation, and convert them into system calls to the browser kernel. Likewise, the interposition layer receives browser kernel’s upcalls, such as keyboard or mouse events, and synthesizes them in the Trident instance.

For example, suppose a user navigates to a web page *a.com*, which embeds a cross-principal frame *b.com*. First, the browser kernel will fetch *a.com*’s HTML content, create a new *a.com* process with a Trident component, and pass the HTML to Trident for rendering. During the rendering process, we intercept the frame navigation event for *b.com*, determine that it is cross-principal, and cancel it. The frame’s DOM element in *a.com*’s DOM is left intact as a placeholder, making the interpo-

Type	Call Name	Description
syscall	getSameOriginContent(URL)	retrieves same origin content
syscall	getCrossOriginContent(URL)	retrieves script or css content
syscall	delegate(URL, delegatedWindowSpec)	delegates screen area to a different principal
syscall	postMessage(windowID, msg, targetOrigin)	cross-frame messaging
syscall	display(windowID, bitmap)	sets the display buffer for the window
syscall	back()	steps back in the window history
syscall	forward()	steps forward in the window history
syscall	navigate (windowID, URL)	navigates a window to URL
syscall	createTopLevelWindow (URL)	creates a new browser tab for the URL specified
syscall	changeWindow (windowID, position, size)	updates the location and size of a window
syscall	writePersistentState (type, state)	allows writing to origin-partitioned storage
syscall	readPersistentState (type)	allows reading of origin-partitioned storage
syscall	lockPersistentState (type)	locks one type of origin-partitioned storage
upcall	destroy(windowID)	closes a browser instance
upcall	resize(windowID, windowSpec)	changes the dimensions of the browser instance
upcall	createPlugin(windowID, URL, content)	creates a plugin instance
upcall	createDocument(windowID, URL, content)	creates a browser instance
upcall	sendEvent(windowID, eventInfo)	passes an event to the browser instance

Table 3: Some Gazelle System Calls

sition transparent to `a.com`. We extract the frame’s position, dimensions, and CSS properties from this element through DOM-related COM interfaces, and send this information in a `delegate` system call to the browser kernel to allow the landlord `a.com` to “rent out” part of its display area to the tenant `b.com`. The browser kernel then creates a new `b.com` process (with a new instance of Trident), and asks it to render `b.com`’s frame. For any rendered display updates for either `a.com` or `b.com`, our interposition code obtains a bitmap of display content from Trident using the `IViewObject` interface and sends it to the browser kernel for rendering.

One intricacy we faced was in rerouting all network requests issued by Trident instances through the browser kernel. We found that interposing on all types of fetches, including frame, script, and image requests, to be very challenging with COM hooks currently exposed by Trident. Instead, our approach relies on a *local web proxy*, which runs alongside the browser kernel. We configure each Trident instance to use our proxy for all network requests, and the proxy converts each request into a corresponding system call to the browser kernel, which then enforces our security policy and completes the request.

One other implementation difficulty that we encountered was to properly manage the layout of cross-origin images. It is easy to render a cross-origin image in a separate process, but difficult to extract the image’s correct layout information from the host page’s Trident instance. We anticipate this to be an overcomable implementation issue. In our current prototype, we are keeping cross-origin images in the same process as their host page for

proper rendering of the pages.

Our interposition layer ensures that our Trident components are never trusted with sensitive operations, such as network access or display rendering. However, if a Trident renderer is compromised, it could bypass our interposition hooks and compromise other principals using the underlying OS’s APIs. To prevent this, we are in the process of implementing an OS-level sandboxing mechanism, which would prevent Trident from directly accessing sensitive OS APIs. The feasibility of such a browser sandbox has already been established in Xax [15] and Native Client [47].

To verify that such an implementation does not cause rendering problems with popular web content, we used our prototype to manually browse through the top 20 Alexa [5] web sites. We checked the correctness of Gazelle’s visual output against unmodified Internet Explorer and briefly verified page interactivity, for example by clicking on links. We found that 19 of 20 web sites rendered correctly. The remaining web site exposed a (fixable) bug in our interposition code, which caused it to load with incorrect layout. Two sites experienced crashes (due to more bugs) when trying to render embedded cross-principal `<iframe>`’s hosting ads. However, the crashes only affected the `<iframe>` processes; the main pages rendered correctly with the exception of small blank spaces in place of the failed `<iframe>`’s. This illustrates a desirable security property of our architecture, which prevents malicious or misbehaving cross-origin tenants from affecting their landlords or other principals.

	Gazelle		Internet Explorer 7		Google Chrome	
	Time	Memory Used	Time	Memory Used	Time	Memory Used
1. Browser startup (no page)	668 ms	9 MB	635 ms	14 MB	500 ms	25 MB
2. New tab (blank page)	602 ms	14 MB	115 ms	0.7 MB	230 ms	1.8 MB
3. New tab (google.com)	939 ms	16 MB	499 ms	1.4 MB	480 ms	7.6 MB
4. Navigate from google.com to google.com/ads	955 ms	6 MB	1139 ms	3.1 MB	1020 ms	1.4 MB
5. Navigate to nytimes.com (with a cross-origin frame)	5773 ms	88 MB	3213 ms	53 MB	3520 ms	19.4 MB

Table 4: Loading times and memory overhead for a sequence of typical browser operations.

8 Evaluation

In this section, we measure the impact of our architecture on browser performance. All tests were performed on an Intel 3.00Ghz Core 2 Duo with 4GB of RAM, running 32-bit Windows Vista with a gigabit Ethernet connection. To evaluate Gazelle’s performance, we measured page loading latencies, the memory footprint, and responsiveness of our prototype in comparison with IE7, a monolithic browser, and Google Chrome v1, a multi-process browser. We found that while Gazelle performs on-par with commercial browsers while browsing within an origin, it introduces some overhead for cross-origin navigation and rendering embedded cross-origin principals (e.g., frames). Nevertheless, our main sources of overhead stem from our interposition layer, various initialization costs for new browser instances, and the un-optimized nature of our prototype. We point out simple optimizations that would eliminate much of the overhead along the way.

Page load latency. Table 4 shows the loading times for a series of browser operations a typical user might perform using our prototype, IE7, and Google Chrome. The operations are repeated one after another within the same browser. A web page’s loading time is defined as the time between pressing the “Go” button and seeing the fully-rendered web page. All operations include network latency.

Operation 1 measures the time to launch the browser and is similar for all three browsers. Although Gazelle’s browser kernel is small and takes only 225 ms to start, Gazelle also initializes the local proxy subsystem (see Section 7), which takes an additional 443 ms. Operations 2 and 3 each carry an overhead of creating a new process in Gazelle and Chrome, but not IE7. Operation 4 reuses the same `google.com` process in Gazelle to render a same-origin page to which the user navigates via a link on `google.com`. Here, Gazelle is slightly faster than both IE7 and Chrome, possibly because Gazelle does not yet manage state such as browsing history between nav-

igations. Finally, operation 5 causes Gazelle to create a new process for `nytimes.com` to render the popular news page³. In addition, NYTimes contains an embedded cross-principal `<iframe>`, which triggers window delegation and another process creation event in Gazelle. Gazelle’s overall page load latency of 5773 ms includes the rendering times of both the main page and the embedded `<iframe>`, with the main page becoming visible and interactive to the user in 5085 ms.

Compared to both IE7 and Chrome, it is expected that Gazelle will have a performance overhead due to extra process creation costs, messaging overhead, and the overhead of our Trident interposition layer as well as Trident itself. Table 5 breaks down the major sources of overhead involved in rendering the three sites in Table 4.

Our Trident interposition layer is a big source of overhead, especially for larger sites like NYTimes.com, where it consumes 813 ms. Although we plan to optimize our use of Trident’s COM interfaces, we are also limited by the Trident host’s implementation of the hooks that we rely on, and by the COM layer which exposes these hooks. Nevertheless, we believe we could mitigate most of this latency if Trident were to provide us with a direct (non-COM) implementation for a small subset of its hooks that Gazelle requires.

Our local proxy implementation for network interposition constitutes another large source of overhead, for example 541 ms for NYTimes.com. Much of this overhead would disappear if Trident were to make direct network system calls to the browser kernel, rather than going through an extra proxy indirection. Another part of this overhead stems from the fact that the browser kernel currently releases web page data only when a whole network transfer finishes; instead, it could provide browser instances with chunks of data as soon as they arrive (e.g., by changing `getContent` system calls to the semantics of a UNIX `read()` system call), allowing them to better overlap network transfers with rendering.

Process creation is an expected source of overhead that

³In contrast, Chrome reuses the tab’s old `google.com` process

increases whenever sites embed cross-principal content, such as NYTimes's cross-origin `<iframe>`. As well, each process must instantiate and initialize a new Trident object, which is expensive. As an optimization, we could use a worker pool of a few processes that have been pre-initialized with Trident. This would save us 275 ms on NYTimes's load time and 134 ms on `google.com`'s load time.

We encountered an unexpected performance hit when initializing named pipes that we use to transfer system calls: a new process's first write to a pipe stalls for a considerable time. This could be caused by initialization of an Interop layer between .NET and the native Win32 pipe interfaces, on which our implementation relies. We can avoid this overhead by either using an alternate implementation of a system call transfer mechanism, or pre-initializing named pipes in our worker pool. This would save us 439 ms in NYTimes's render time.

Retrieving bitmap display updates from Trident and sending them to the browser kernel is expensive for large, complex sites such as NYTimes.com, where this takes 422 ms. Numerous optimizations are possible, including image compression, VNC-like selective transfers, and a more efficient bitmap sharing channel between Trident and the browser kernel. Our mechanism for transferring bitmap updates currently performs an inefficient .NET-based serialization of the image's data (which takes 176 ms for NYTimes); passing this data directly would further improve performance.

Overall, we believe that with the above optimizations, Gazelle's performance would be on par with production browsers like Chrome or IE8; for example, we anticipate that NYTimes.com could be rendered in about 3.6 s.

Memory overhead. As a baseline measurement, the browser kernel occupies around 9MB of memory after a page load. This includes the user interface components of the browser to present the rendered page to the user and the buffers allocated for displaying the rendered page. Memory measurements do not include shared libraries used by multiple processes.

Table 4 shows the amount of memory for performing various browsing operations. For example, to open a new tab to a blank page, Gazelle consumes 14MB, and to open a new tab for `google.com`, Gazelle consumes an additional 16MB. Each empty browser instance uses 1.5MB of internal storage plus the memory required for rendered content. Given our implementation, the latter closely corresponds to Trident's memory footprint, which at the minimum consists of 14MB for a blank page. In the case of NYTimes, our memory footprint further increases because of structures allocated by the interposition layer, such as a local DOM cache.

Responsiveness. We evaluated the response time of a user-generated event, such as a mouse click. When the

browser kernel detects a user event, it issues a `sendEvent` upcall to the destination principal's browser instance. Such calls take only 2 ms on average to transfer, plus 1 ms to synthesize in Trident. User actions might lead to display updates; for example, a display update for `google.com` would incur an additional 77 ms. Most users should not perceive this overhead and will experience good responsiveness.

Process creation. In addition to latency and memory measurements we also have tested our prototype on the top 100 popular sites reported by Alexa [5] to provide an estimate of the number of processes created for different sites. Here, we place a cross-origin image into a separate process to evaluate our design. The number of processes created is determined by the use of different-origin content on sites, which is most commonly image content. For the top 100 sites, the median number of processes required to view a single page is 4, the minimum is 1, and the maximum is 28 (caused by `skyrock.com`, which uses an image farm). Although creation of many processes introduces additional latency and memory footprint, we did not experience difficulties when Gazelle created many processes during normal browsing. Our test machine easily handles a hundred running processes, which are enough to keep 25 average web sites open simultaneously.

9 Discussions on compatibility vs. security

While Gazelle's architecture can be made fully backward compatible with today's web, it is interesting to investigate the compatibility cost of eliminating the insecure policies in today's browsers. We have considered several policies that differ from today's browsers but offer better security. We conducted a preliminary study on their compatibility cost. This is by no means a conclusive or complete study, but only a first look on the topic.

We mostly used the data set of the front pages of the top 100 most popular web sites ranked by Alexa [5]. We used a combination of browser instrumentation with automatic script execution and manual inspection in our study. We consider any visual differences in the rendering of a web page to be a violation of compatibility. We discuss our findings below.

Subdomain treatment Existing browsers and SOP make exceptions for subdomains (e.g., `news.google.com` is a subdomain of `google.com`) [39]: a page can set the `document.domain` property to suffixes of its domain and assume that identity. This feature was one of the few methods for cross-origin frames to communicate before the advent of `postMessage` [25]. Changing `document.domain` is a dangerous practice and violates the Principle of Least Privilege: Once a subdomain sets its domain to a suffix, it has no control over which other

Location	Overhead	Latency		
		blank site	google.com	nytimes.com
	Overhead before rendering			
Browser kernel	- process creation	44 ms	40 ms	78 ms
Browser instance	- creating interposed instances of Trident	94 ms	94 ms	197 ms
Browser instance	- named pipe initialization	137 ms	145 ms	439 ms
	Overhead during rendering			
Browser instance	- proxy-based network interposition	4 ms	134 ms	541 ms
Browser instance	- other Trident interposition	127 ms	122 ms	813 ms
	Overhead after rendering			
Browser instance	- bitmap capture	13 ms	35 ms	196 ms
Browser instance	- bitmap transfer	37 ms	67 ms	226 ms
Browser kernel	- display rendering	10 ms	11 ms	101 ms

Table 5: A breakdown of Gazelle’s overheads involved in page rendering. Note that nytimes.com creates *two* processes for itself and an `<iframe>`; the other two sites create one process.

subdomains can access it. This is also observed by Zalewski [48]. Therefore, it would be more secure not to allow a subdomain to set `document.domain`.

Our experiments indicate that six of the top 100 Alexa sites set `document.domain` to a different origin, though restricting write access to `document.domain` might not actually break the operation of these web sites.

Mixed HTTPS and HTTP Content. When an HTTPS site embeds HTTP content, browsers typically warn users about the mixed content, since the HTTPS site’s content can resist a network attacker, but the embedded HTTP content could be compromised by a network attacker.

When an HTTPS site embeds other HTTP principals (through `<iframe>`, `<object>`, etc.), HTTPS principals and HTTP principals will have different protection domains and will not interfere with each other.

However, when an HTTPS site embeds a script or style sheet delivered with HTTP, existing browsers would allow the script to run with the HTTPS site’s privileges (after the user ignores the mixed content warning). This is dangerous because a network attacker can then compromise the HTTP-transmitted script and attack the HTTPS principal despite its intent of preventing network attackers. Therefore, a more secure policy is to deny rendering of HTTP-transmitted scripts or style sheets for an HTTPS principal. Instead of the Alexa top 100, we identified a few different sites that provide SSL sessions for parts of their web application: *amazon.com*, *mail.google.com*, *mail.microsoft.com*, *blogger.com*, and a few popular banking sites where we have existing accounts. This allows us to complete the login process during testing. These sites do not violate this policy. In addition, we have also gathered data from one of the author’s browsing sessions over the course of a few months and found that out of 5,500 unique SSL URLs seen, less

than two percent include HTTP scripts and CSS.

Layout policies. The opaque overlay policy allows only opaque (and not transparent) cross-origin frames or objects (Section 5.2). We test this policy with the top 100 Alexa sites by determining if any cross-origin frames or objects are overlapped with one another. We found that two out of 100 sites attempt to violate this policy. This policy does not generate rendering errors; instead, we convert transparent cross-origin elements to opaque elements when displaying content.

We also tested the 2D display delegation policy that we analyzed in Section 5.2. We found this policy to have higher compatibility cost than our opaque overlay policy: six of the top 100 sites attempt to violate this policy.

Sites that attempt to violate either policy have reduced functionality, and will render differently than what the web page author intends.

Plugins. Existing plugin software must be adapted (ported or binary-rewritten) to use browser kernel system calls to accomplish its tasks. Of top 100 Alexa sites, 34 sites use Flash, but no sites use any other kinds of plugins. This indicates that porting or adapting Flash alone can address a significant portion of the plugin compatibility issue.

10 Concluding Remarks

We have presented Gazelle, the first web browser that qualifies as a multi-principal OS for web site principals. This is because Gazelle’s browser kernel *exclusively* manages resource protection, unlike all existing browsers which allow cross-principal protection logic to reside in the principal space. Gazelle enjoys the security and robustness benefit of a multi-principal OS: a compromise or failure of one principal leaves other principals and the browser kernel intact.

Our browser construction exposes challenging design issues that were not seen in previous work, such as providing legacy protection to cross-origin script source and cross-principal, cross-process display and event protection. We are the first to provide comprehensive solutions to them.

The implementation and evaluation of our IE-based prototype shows promise of a practical multi-principal OS-based browser in the real world.

In our future work, we are exploring the fair sharing of resources among web site principals in our browser kernel and a more in-depth study of the tradeoffs between compatibility and security in browser policy design.

11 Acknowledgements

We thank Spencer Low, David Ross, and Zhenbin Xu for giving us constant help and fruitful discussions. We thank Adam Barth and Charlie Reis for their detailed and insightful feedback on our paper. We also thank the following folks for their help: Barry Bond, Jeremy Condit, Rich Draves, David Driver, Jeremy Elson, Xiaofeng Fan, Manuel Fandrich, Cedric Fournet, Chris Hawblitzel, Jon Howell, Galen Hunt, Eric Lawrence, Jay Lorch, Rico Malvar, Wolfram Schulte, David Wagner, Chris Wilson, and Brian Zill. We also thank our paper shepherd Niels Provos for his feedback over our last revisions.

References

- [1] Changes in allowScriptAccess default (Flash Player). <http://www.adobe.com/go/kb403183>.
- [2] Developer center: Security changes in Flash Player 7. http://www.adobe.com/devnet/flash/articles/fplayer_security.html.
- [3] Security advisories for Firefox 2.0. <http://www.mozilla.org/security/known-vulnerabilities/firefox20.html>.
- [4] .NET Framework Developer Center, 2008. <http://msdn.microsoft.com/en-us/netframework/default.aspx>.
- [5] Alexa, 2009. <http://www.alexa.com/>.
- [6] M. Barnett, K. Rustan, M. Leino, and W. Schulte. The Spec# programming system: An overview. In LNCS, editor, *CAS-SIS*, volume 3362. Springer, 2004. <http://research.microsoft.com/en-us/projects/specsharp/>.
- [7] A. Barth and C. Jackson. Protecting browsers from frame hijacking attacks, April 2008. <http://crypto.stanford.edu/websec/frames/navigation/>.
- [8] A. Barth, C. Jackson, and J. C. Mitchell. Securing frame communication in browsers. In *In Proceedings of the 17th USENIX Security Symposium (USENIX Security)*, 2008.
- [9] A. Barth, C. Jackson, C. Reis, and T. G. C. Team. The security architecture of the Chromium browser, 2008. <http://crypto.stanford.edu/websec/chromium/chromium-security-architecture.pdf>.
- [10] S. Chen, D. Ross, and Y.-M. Wang. An Analysis of Browser Domain-Isolation Bugs and A Light-Weight Transparent Defense Mechanism. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2007.
- [11] R. S. Cox, J. G. Hansen, S. D. Gribble, and H. M. Levy. A Safety-Oriented Platform for Web Applications. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2006.
- [12] D. Crockford. JSONRequest. <http://www.json.org/jsonrequest.html>.
- [13] D. Crockford. The Module Tag: A Proposed Solution to the Mashup Security Problem. <http://www.json.org/module.html>.
- [14] Document Object Model. <http://www.w3.org/DOM/>.
- [15] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch. Leveraging legacy code to deploy desktop applications on the web. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, 2008.
- [16] Firefox 3 for developers, 2008. https://developer.mozilla.org/en/Firefox_3_for_developers.
- [17] Mozilla Browser and Mozilla Firefox Remote Window Hijacking Vulnerability, 2004. <http://www.securityfocus.com/bid/11854/>.
- [18] Security Advisories for Firefox 2.0. <http://www.mozilla.org/security/known-vulnerabilities/firefox20.html>.
- [19] D. Flanagan. *JavaScript: The Definitive Guide*. O'Reilly Media Inc., August 2006.
- [20] Adobe Flash Player 9 Security, July 2008. http://www.adobe.com/devnet/flashplayer/articles/flash_player_9_security.pdf.
- [21] C. Grier, S. Tang, and S. T. King. Secure web browsing with the OP web browser. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, 2008.
- [22] J. Grossman. Advanced Web Attack Techniques using GMail. <http://jeremiahgrossman.blogspot.com/2006/01/advanced-web-attack-techniques-using.html>.
- [23] W. H. A. T. W. Group. Web Applications 1.0, February 2007. <http://www.whatwg.org/specs/web-apps/current-work/>.
- [24] HTML 5 Editor's Draft, October 2008. <http://www.w3.org/html/wg/html5/>.
- [25] What's New in Internet Explorer 8, 2008. <http://msdn.microsoft.com/en-us/library/cc288472.aspx>.
- [26] Microsoft Internet Explorer Remote Window Hijacking Vulnerability, 2004. <http://www.securityfocus.com/bid/11855>.
- [27] S. Ioannidis and S. M. Bellovin. Building a secure web browser. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, 2001.
- [28] S. Ioannidis, S. M. Bellovin, and J. M. Smith. Sub-operating systems: a new approach to application security. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 108–115, New York, NY, USA, 2002. ACM.
- [29] C. Jackson and A. Barth. Beware of Finer-Grained Origins. In *Web 2.0 Security and Privacy*, May 2008.
- [30] C. Jackson, A. Barth, A. Bortz, W. Shao, and D. Boneh. Protecting Browsers from DNS Rebinding Attacks. In *Proceedings of ACM Conference on Computer and Communications Security*, 2007.

- [31] JavaScript Object Notation (JSON). <http://www.json.org/>.
- [32] D. Kristol and L. Montulli. HTTP State Management Mechanism. IETF RFC 2965, October 2000.
- [33] T. W. Mathers and S. P. Genoway. *Windows NT Thin Client Solutions: Implementing Terminal Server and Citrix MetaFrame*. Macmillan Technical Publishing, Indianapolis, IN, November 1998.
- [34] IEBlog: IE8 Security Part V: Comprehensive Protection, 2008. <http://blogs.msdn.com/ie/archive/2008/07/02/ie8-security-part-v-comprehensive-protection.aspx>.
- [35] Microsoft security bulletin. <http://www.microsoft.com/technet/security/>.
- [36] Microsoft Security Intelligence Report, Volume 5, 2008. <http://www.microsoft.com/security/portal/sir.aspx>.
- [37] C. Reis and S. D. Gribble. Isolating web programs in modern browser architectures. In *Proceedings of Eurosys*, 2009.
- [38] T. Richardson, Q. Stafford-Fraser, K. R. Wood, and A. Hopper. Virtual network computing. *IEEE Internet Computing*, 2(1):33–38, 1998.
- [39] J. Ruderman. The Same Origin Policy. <http://www.mozilla.org/projects/security/components/same-origin.html>.
- [40] R. W. Scheifler and J. Gettys. The X window system. *ACM Transactions on Graphics (TOG)*, 5(2):79–109, April 1986.
- [41] J. S. Shapiro, J. Vanderburgh, E. Northup, and D. Chizmadia. Design of the EROS TrustedWindow system. In *Usenix Security*, 2004.
- [42] Symantec Global Internet Security Threat Report: Trends for July - December 07, April 2008.
- [43] H. J. Wang, X. Fan, J. Howell, and C. Jackson. Protection and Communication Abstractions in MashupOS. In *ACM Symposium on Operating System Principles*, October 2007.
- [44] Cross-Domain Vulnerability In Microsoft Internet Explorer 6. <http://cyberinsecure.com/cross-domain-vulnerability-in-microsoft-internet-explorer-6/>.
- [45] The XMLHttpRequest Object. <http://www.w3.org/TR/XMLHttpRequest/>.
- [46] W3C XMLHttpRequest Level 2. <http://dev.w3.org/2006/webapi/XMLHttpRequest-2/>.
- [47] B. Yee, D. Sehr, G. Dardyk, B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2009.
- [48] M. Zalewski. Browser security handbook, 2008. <http://code.google.com/p/browsersec/wiki/Main>.

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of system administrators, developers, programmers, and engineers working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering technical excellence and innovation
- encouraging computing outreach in the community at large
- providing a neutral forum for the discussion of critical issues

For more information about membership and its benefits, conferences, or publications, see <http://www.usenix.org>.

SAGE, a USENIX Special Interest Group

SAGE is a Special Interest Group of the USENIX Association. Its goal is to serve the system administration community by:

- Establishing standards of professional excellence and recognizing those who attain them
- Promoting activities that advance the state of the art or the community
- Providing tools, information, and services to assist system administrators and their organizations
- Offering conferences and training to enhance the technical and managerial capabilities of members of the profession

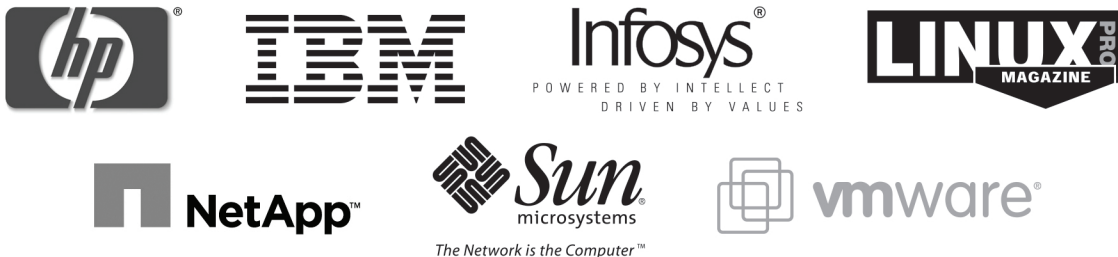
Find out more about SAGE at <http://www.sage.org>.

Thanks to USENIX & SAGE Corporate Supporters

USENIX Patrons



USENIX Benefactors



USENIX & SAGE Partners

Ajava Systems, Inc.
DigiCert® SSL Certification
FOTO SEARCH Stock Footage and
Stock Photography
Hyperic Systems Monitoring
Splunk
Zenoss

USENIX Partners

Cambridge Computer Services, Inc.
GroundWork Open Source
Solutions
Xirrus

SAGE Partner

MSB Associates

ISBN-13: 978-1-931971-69-0

90000



9 781931 971690